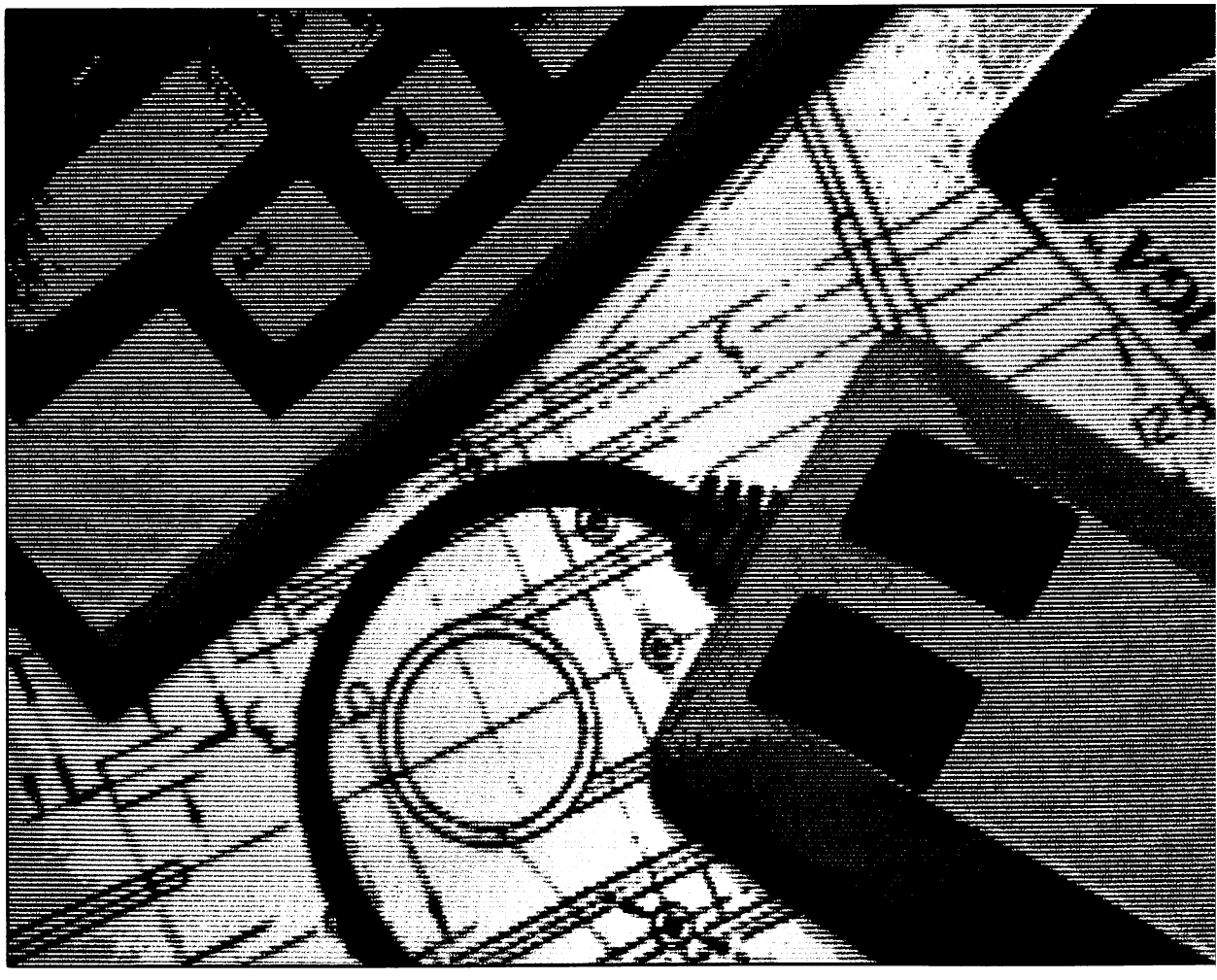
AMIGA™



Lattice C Compiler Manual

Lattice C Compiler
Manual

PRINTED in U.S.A.

CBM Product Number 327270-01 rev 1.0 8.28.85

TABLE OF CONTENTS

SECTION 1:
Introduction and Basic Concepts

This document provides a functional description of an implementation of the Lattice C compiler, a portable compiler for the high level programming language called C. It makes no attempt to discuss either programming fundamentals or how to program in C itself. Although it has become somewhat outdated by recent enhancements to the language, the authoritative text on C remains The C Programming Language, by Brian W. Kernighan and Dennis M. Ritchie (Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978); it provides an excellent tutorial introduction to the language.

1.1 Documentation

The manual is divided into six sections. The first three address the Lattice implementation in general terms, while the second three describe the details of program development for a particular context. This context can be characterized by three variables: (1) the host operating system, under which the compiler executes and programs are developed; (2) the target processor, which will execute the machine language instructions generated by the compiler; and (3) the target operating system, under which developed programs will be loaded for execution. In some contexts the latter may be null, that is, no operating system is involved because programs are part of a stand-alone system and are placed in read-only memory. If the host and target processors and operating systems are all the same, the compiler is termed a native compiler, while if they differ, it is called a cross compiler.

The general topics of the sections of the manual are as follows. First, this introduction presents some basic concepts. Second, the language accepted by the compiler is described. The third section then presents the standard library functions in functional groups with usage summaries. The fourth section describes program development, including operating instructions for the compiler on the host operating system, while the fifth section describes the generated code and run-time program environment for the target processor. The sixth section describes the target operating system interfaces in terms of the standard library functions and the special functions provided for that system.

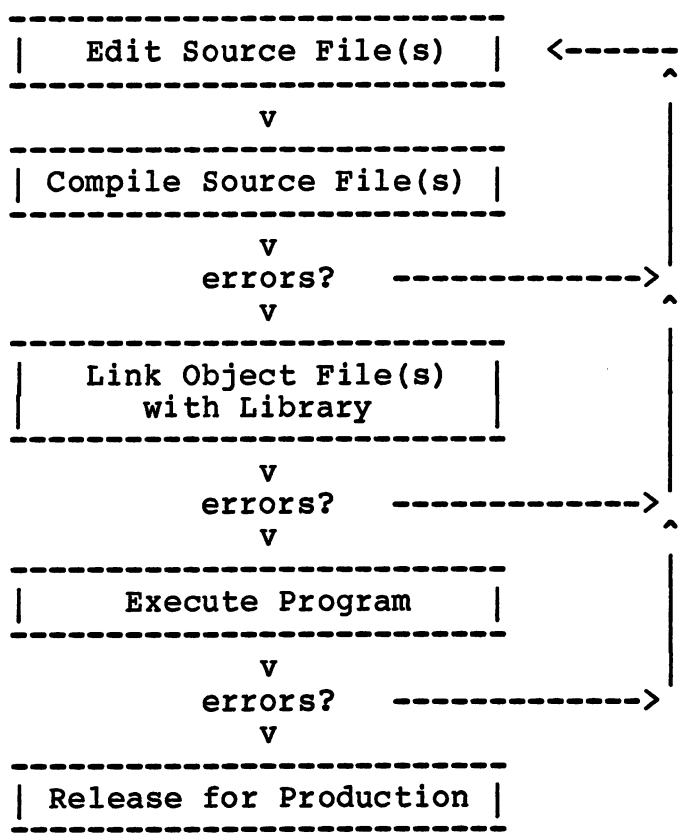
As this document is intended to serve as a reference manual, each topic is usually presented in full technical detail as it is encountered. Some reference to sections not yet encountered is unavoidable, but these references are specifically noted.

Error messages are described in Appendix A, and reporting proce-

dures for compiler errors in Appendix B. A list of the files shipped with this version of Lattice C are contained in Appendix C.

1.2 Basic Concepts

Lattice C is a "classic" implementation of a compiled programming language, whose goal is fast, compact programs suitable for production use. Program development in such a system consists of a number of iterated logical steps, as the diagram below shows:



After each step in the process, a check for errors is made. Notice that the detection of an error invariably leads back to the first step, and the repetition of the sequence.

(1) Source file preparation

The first step is to prepare text files for submission to the compiler. These files define **functions**, which determine the flow of control in the program, and **data elements**, which are used in the program for the storage and manipulation of data. Since program source files are ordinary text files, any general purpose text file editor may be used to create and revise these files.

(2) Object module compilation

The function of the compiler is to translate the source language representation of the program's components into what is called object format. This format is an encoded sequence of machine language instructions with linkage information. It represents a form intermediate between the source files and the final program file. Along with the translation process, the compiler verifies that the source file conforms to the legal specifications of the language. An error-free compilation does not, however, insure a working program.

(3) Program linking

In order to make programming a more manageable task, programs can be built out of collections of smaller components called modules. The collection of the component object modules into a single program is called program linking, and it requires that all of the modules are compiled and available for inclusion in the finished program. Programs can also incorporate modules from a library of useful functions provided with the compiler.

(4) Program execution and debugging

Once the program is completely linked, it can be executed to perform the desired processing. Since most programs rarely work exactly as designed, the detection and correction of program errors becomes necessary. This process is known as debugging, and involves repetition of the whole process of editing, compiling, linking and execution.

Because of the number of steps involved in the process and the length of time required to perform each step, this kind of program development tends to be costly as far as the time required to produce finished programs. The end product, however, is a program which executes quickly and requires a minimal amount of memory for execution.

SECTION 2:
Language Definition

The Lattice portable C compiler accepts a program written in the C programming language, determines the elementary actions specified by that program, and eventually translates those actions into machine language instructions. Although the final result of these processes is highly machine-dependent, the actual language accepted by the compiler is, for the most part, independent of any system or implementation details. This section presents the language defined by Version 3 of the Lattice C compiler. Since this language conforms closely to that of UNIX System V and differs in several ways from previous implementations of Lattice C, the major differences are first presented. The major features of the language are then discussed, not in any attempt at completeness, but simply for the sake of showing them from a different perspective.

This version of the Lattice C language definition still uses the Kernighan and Ritchie text The C Programming Language as a base for comparisons; in the future, a complete language reference for Lattice C will be issued as Section 2. Until that time, the specification of the language will remain the Kernighan and Ritchie "C Reference Manual" as amended in Section 2.3. For those programmers who are unfamiliar with some of the features of the lanugage added after the publication of the Kernighan and Ritchie text, some examples have been provided in Section 2.4.

2.1 Summary of Differences

There are two classes of differences that appear in a discussion of an implementation of a programming language. The first class is that of actual semantic differences, that is, variations which cause the meaning of language constructs to differ. The second class is merely a reflection of the practical limitations to which all programs -- including compilers -- are subject. Each of the following subsections presents the respective details for the Lattice implementation of C.

2.1.1 Differences from Previous Versions

Version 3 of Lattice C incorporates several of the more recent additions to the C programming language, as it evolved at Bell Telephone Laboratories. It conforms closely to the version of the language supported under UNIX System V. Efforts are now under way to define an ANSI standard for the language, but many of the features being discussed by the standardization committee are not yet firmly established. Accordingly, Lattice has not chosen to support any of the new language features with one exception: an argument type-checking mechanism has been provided which, although not fully compatible with the proposed standard,

at least addresses one of the more difficult debugging problems of the language. For programmers who have used a previous version of Lattice C, here is a summary of the major changes in Version 3:

- The reserved symbol **unsigned** may be applied to any of the integral data types, i.e., it is now a true modifier rather than a separate data type.

- The special type **void** is supported; it describes a function which has no return value.

- Enumerated data types may be declared using the special type **enum**; these objects are integral but may only assume values from a specified list of identifiers. The declaration is similar to that for structure and union objects; the identifiers defining permissible values can be used an integer constants.

- Structures and unions of identical type may be copied by means of an assignment statement. Structures and unions may be passed by value to functions, and functions may return a structure or union as a function value.

- External functions may be declared with a list of type names enclosed in the function call parentheses. When such a function is called, the compiler checks the supplied arguments against the expected types and number of parameters. Warning messages are issued if the actual types do not match.

- Several minor deviations from the standard language have been corrected. The compiler no longer accepts the character $ in identifiers; comments do not nest; multiple character constants are not accepted; and separate copies of string constants are generated for each instance of identical strings. All of these features can be forced to revert to their previous characteristics by means of a compile time option. In addition, **char** declarations can be forced to be interpreted as **unsigned char**. See Section 4.1.1 for details.

Several new warning messages have been added to the compiler as an assistance to debugging, including a warning if an auto variable is used in a context where it may not have been initialized, and a warning if a locally visible object is declared but never referenced. In addition, the error recovery of the compiler has been improved in an effort to reduce the number of spurious error messages.

2.1.2 Arbitrary Limitations

Although the definition of a programming language is an idealized abstraction, any real implementation is constrained by a number of factors, not the least of which is practicality. The Lattice compiler imposes the following arbitrary restrictions on the language it accepts:

- The maximum value of the constant expression defining the size of a single subscript of an array is two less than the largest unsigned int (65533 for a 16-bit int).
- The maximum length of an input source line is 256 bytes.
- The maximum size of a string constant is 256 bytes.
- Macros with arguments are limited to a maximum number of 8 arguments.
- The maximum length of the substitution text for a #define macro is 256 bytes.
- The maximum level of #include file nesting is 10.

These limitations are imposed because of the way objects are represented internally by the compiler; our hope is that they are reasonably large enough for most real programs.

2.2 Major Language Features

The material presented in this section is meant to clarify some of the language features which are not always fully defined. These are features which depend on implementation decisions made in the design of the compiler itself, or on interpretations of the language definition. Those language features which are specifically machine dependent are described in Section 5 of this manual.

2.2.1 Pre-processor Features

The Lattice C compiler supports the full set of pre-processor commands described in Kernighan and Ritchie. Most implementations perform the pre-processor commands concurrently with lexical and syntactic analysis of the source file, because an additional compilation step can be avoided by this technique. Other versions of the compiler incorporate a separate pre-processor phase in order to reduce the size of the first phases of the compiler. In either case, the analysis of the pre-processor commands is largely independent of the compiler's C language analysis. Thus, #define text substitutions are not generally performed for any of the pre-processor commands, although nesting of macro definitions is possible since

substituted text is always re-scanned for new #define symbols.

Because the compiler uses a text buffer of fixed size, a particularly complex macro may occasionally cause a line buffer overflow condition; usually, however, this error occurs when more than one macro reference occurs in the same source line, and can be circumvented by placing the macros on different lines.

Circular definitions such as

```
#define A B
#define B A
```

will be detected by the compiler if either A or B is ever used, as will more subtle loops. Like many other implementations of C, the Lattice compiler supports nested macro definitions, so that if the line

```
#define XYZ 12
```

is followed later by

```
#define XYZ 43
```

the new definition takes effect, but the old one is not forgotten. Note that the compiler issues a warning message whenever a redefinition occurs. In other words, after encountering

```
#undef XYZ
```

the former definition (12) is restored. To completely undefine XYZ, an additional #undef is required. The rule is that each #define must be matched by a corresponding #undef before the symbol is truly "forgotten".

Two clarifications should be noted with regard to the #if command. If a symbol appears in an #if expression which has not been defined in a #define command, it is interpreted as if a value of zero had been specified. Similarly, a symbol which has been defined with a null substitution text is interpreted as if a value of one had been specified. These conventions are consistent with #ifdef usage, and permit the use of expressions like

```
#if SYM1 | SYM2 | SYM3
```

which causes subsequent code to be processed if any of the symbols are defined.

2.2.2 Arithmetic Objects

Five types of arithmetic objects are supported by the Lattice compiler; along with pointers, these objects represent the entities which can be manipulated in a C program. The types are:

short or short int
char
long or long int
float
double or long float

In addition, unsigned may be applied as a modifier to any of the integral data types. This modification does not affect the size of the object, and is significant only when such an object is used in an expression involving conversion, multiplication, division, comparison, or bitwise right shifts.

The natural size of integers for the target machine (the machine for which code is being generated) is indicated by a plain int type specifier; this type will be identical to either short or long, depending on the architecture of the target machine. Enumeration data types are a special type of integral data; they are treated as int, although the compiler may elect to use a smaller integer for the actual storage if the range of values permits.

The compiler follows the standard pattern for conversions between the various arithmetic types, the so-called "usual arithmetic conversions" described in the Kernighan and Ritchie text. The only exception to this occurs in connection with byte-oriented machines, where expansion of char to int may be avoided if both operands in an expression are char, and the target machine supports byte-mode arithmetic and logical operations.

2.2.3 Derived Objects

The Lattice C compiler supports the standard extensions leading to various kinds of derived objects, including pointers, functions, arrays, and structures and unions. Declarations of these types may be arbitrarily complex, although not all declarations result in a legal object. For example, arrays of functions or functions returning arrays are illegal. The compiler checks for these kinds of declarations and also verifies that structures or unions do not contain instances of themselves. Objects which are declared as arrays cannot have an array length of zero, unless they are formal parameters or are declared extern (see Section 2.2.4). All pointers are assumed to be the same size -- usually, that of a plain int -- with one exception. On word-addressed machines, pointers which point to objects which can appear on an address boundary within a machine word may differ in size from pointers to objects which must be machine-word-aligned.

Note that the size of aggregates (arrays and structures) may be affected by alignment requirements. For example, the array

```
struct {
    short i;
    char c;
    } x[10];
```

will occupy 40 bytes on machines which require short objects to be aligned on an even byte address. Any such alignment padding will be included in the value returned by sizeof.

2.2.4 Storage Classes

Declared objects are assigned by the compiler to storage offsets which are relative to one of several different storage bases. The assigned storage base depends on the explicit storage class specified in the declaration, or on the context of the declaration, as follows:

External An object is classified as external if the extern keyword is present in its declaration, and the object is not later defined in the source file (that is, it is not declared outside the body of any function without the extern keyword). Storage is not allocated for external items because they are assumed to exist in some other file, and must be included during the linking process that builds a set of object modules into a load module.

Static An object is classified as static if the static keyword is present in its declaration or if it is declared outside the body of any function without an explicit storage class specifier. Storage is allocated for static items in the data section of the object module; all such locations are initialized to zero unless an initializer expression is included in the declaration (see Section 2.2.6). Static items declared outside the body of any function without the static keyword are visible in other files, that is, they are externally defined. Note that string constants are allocated as static items, and are treated as unnamed static arrays of char.

Auto An object is classified as auto if the auto keyword is present in its declaration, or if it is declared inside the body of any function without an explicit storage class specifier (it is illegal to declare an object auto outside the body of a function).

Storage is presumably allocated for auto items using a stack mechanism during execution of the function in which they are defined.

Formal An object is classified as formal if it is a formal parameter to one of the functions in the source file. Storage is presumably allocated for formal items when a function call is made during execution of the program.

Note that the first phase of the compiler makes no assumption about the validity of the register storage class declarator. Items which are declared register are so flagged, but storage is allocated for them anyway against either the auto or the formal storage base. The implementation of register is machine-dependent and may not be supported in some cases.

Note also that if the -x compile-time option is used, the implicit storage class for items declared outside the body of any function changes from static to extern. This allows a single header file to be used for all external data definitions. When the main function is compiled, the -x option is not used, and so the various objects are defined and made externally visible; when the other functions are compiled the -x option causes the same declarations to be interpreted as references to objects defined elsewhere.

2.2.5 Scope of Identifiers

The Lattice compiler conforms almost exactly to the scope rules discussed in Appendix A of the Kernighan and Ritchie text (pp. 205-206). The only exception arises in connection with structure and union member names, where (in accordance with later versions of the language) the compiler keeps separate lists of member names for each structure or union; this means that additional classes of non-conflicting identifiers occur for the various structures and unions. Two additional points are worth clarifying.

First, when identifiers are declared at the beginning of a statement block internal to a function (other than the first block immediately following the function name), storage for any auto items declared is allocated against the current base of auto storage. When the statement block terminates, the next available auto storage offset is reset to its value preceding those declarations. Thus, that storage space may be reused by later local declarations. Rather than generate explicit allocate and deallocate operations, the compiler uses this mechanism to compute the total auto storage required by the function; the resulting storage is allocated whenever the function is called. With this scheme, functions will allocate possibly more storage than will be needed (in the event that those inner statement

blocks are not executed), but the need for run-time dynamic allocation within the function is avoided.

Second, when an identifier with a previous declaration is redefined locally in a statement block with the **extern** storage class specifier, the previous definition is superseded in the normal fashion but the compiler also verifies compatibility with any preceding extern definitions of the same name. This is done in accordance with the principle expressed in the text, namely that all functions in a given program which refer to the same external identifier refer to the same object. Within a source file, the compiler also verifies that all external declarations agree in type. The point is that in this particular case -- where a local block redefines an identifier as extern -- the declaration effectively does not disappear upon termination of the block, since the compiler now has an additional external item for which it must verify equivalent declarations.

2.2.6 Initializers

Objects which are of the **static** storage class (as defined in Section 2.2.4) are guaranteed to contain binary zeros when the program begins execution, unless an initializer expression is used to define a different initial value. The Lattice compiler supports the full range of initializer expressions described in Kernighan and Ritchie, but restricts the initialization of pointers somewhat. An arithmetic object may be initialized with an expression that evaluates to an arithmetic constant which, if not of the appropriate type, is converted to that of the target object.

The expression used to initialize a pointer is more restricted: it must evaluate to the **int** constant zero or to a pointer expression yielding a pointer of exactly the same type as the pointer being initialized. This pointer expression can include the address of a previously declared **static** or **extern** object, plus or minus an **int** constant, but it cannot incorporate a cast (type conversion) operator, because pointer conversions are not evaluated at compile time (exception: a cast operator can be used on an **int** constant but not on a variable name). This restriction makes it impossible to initialize a pointer to an array unless the & operator is allowed to be used on an array name, because the array name without the preceding & is automatically converted to a pointer to the first element of the array. Accordingly, the Lattice compiler accepts the & operator on an array name so that declarations such as

```
int a[5], (*pa)[5] = &a;
```

can be made. Note that if a pointer to a structure (or union) is being initialized, the structure name used to generate an address must be preceded by the & operator.

More complex objects (arrays and structures) may be initialized by bracketed, comma-separated lists of initializer expressions, with each expression corresponding to an arithmetic or pointer element of the aggregate. A closing brace can be used to terminate the list early; see Appendix A of Kernighan and Ritchie for examples. Unions may be initialized in this implementation; a single expression corresponding to the first member of the union is used. A character array may be initialized with a string constant which need not be enclosed in braces; this is the only exception to the rule requiring braces around the list of initializers for an aggregate.

Initializer expressions for auto objects can only be applied to simple arithmetic or pointer types (not to aggregates), and are entirely equivalent to assignment statements.

2.2.7 Expression Evaluation

All of the standard operators are supported by the Lattice compiler, in the standard order of precedence (see p. 49 of Kernighan and Ritchie). Expressions are evaluated using an operator precedence parsing technique which reduces complex expressions to a sequence of unary and binary operations involving at most two operands. Operations involving only constant operands (including floating point constants) are evaluated by the compiler immediately, but no special effort is made to re-order operands in order to group constants. Thus, expressions such as

```
c - 'A' + 'a'
```

must be parenthesized so that the compiler can evaluate the constant part:

```
c + ('a' - 'A')
```

If at least one operand in an operation is not constant, the intermediate expression result is represented by a temporary storage location, known as a temporary. The temporary is then "plugged into" the larger expression and becomes an operand of another binary or unary operation; the process continues until the entire expression has been evaluated. The lifetimes of temporaries and their assignment to storage locations are determined by a subroutine internal to the first phase of the compiler, which recognizes identically generated temporaries within a straight-line block of code and eliminates recomputation of equivalent results. Thus, common sub-expressions are recognized and evaluated only once. For example, in the statement

```
a[i+1] = b[i+1];
```

the expression i+1 will be evaluated once and used for both subscripting operations. Expressions which produce a result that is never used and which have no side effects, such as

```
i+j;
```

are discarded by this same subroutine.

Within the block of code examined by the temporary analysis subroutine, operations which produce a temporary result are noted and remembered so that later equivalent operations may be deleted, as noted above. Two conditions (other than function calls, which may have undetermined side effects) cause the subroutine to discard an operation and no longer check for the equivalent operation later: (1) if either of its operands appears directly as a result of a subsequent operation; or (2) if a subsequent operation defines an indirect (i.e., through a pointer) result for the same type of object as one of the original operands. The latter condition is based on the compiler's assumption that pointers are always used to refer to the correct type of target object, so that, for example, if an assignment is made using an **int** pointer only objects of type int can be changed. Only when the programmer indulges in <u>type punning</u> -- using a pointer to inspect an object as if it were a different type -- is this assumption invalid, and it is hard to conceive of a case where the common sub-expression detection will cause a problem with this somewhat dubious practice. Such inspections are generally better left to assembly language modules in any case.

With the exception of this common sub-expression detection, which may replace an operation with a previous, equivalent one, expressions are evaluated in strict left-to-right order as they are encountered, except, of course, where that is prevented by operator precedence or parentheses. It is best not to make any assumptions, however, about the order of evaluation, since the code generation phase is generally free to re-order the sequence of many operations. The most important exceptions are the logical OR (||) and logical AND (&&) operators, for which the language definition guarantees left-to-right evaluation. The code generation phase may have other effects on expression evaluation. Check the implementation section of this manual for full details.

2.2.8 Control Flow

C offers a rich set of statement flow constructs, and the Lattice compiler supports the full complement of them. Some minor points of clarification are noted here. First of all, the compiler does verify that switch statements contain (1) at least one case or default entry; (2) no duplicate case values; and (3)

not more than one default entry. Code within a switch statement which cannot be reached is flagged as an error by the compiler.

The code generation phase generally makes a special effort to generate efficient sequences for control flow. In particular, the size and number of branch instructions is kept to a minimum by extensive analysis of the flow within a function, unreachable code is discarded, and switch statements are analyzed to determine the most efficient of several possible machine language constructs. Check the implementation section of this manual for the details regarding this particular code generator.

2.3 Comparison to the Kernighan & Ritchie "C Reference Manual"

The most precise definition of the C programming language generally available is in Appendix A of the Kernighan and Ritchie text, which is entitled C Reference Manual. This section presents, in the same order defined in the text, a series of amendments or annotations to that manual; this commentary explicitly states any deviations of the Lattice C language implementation from the features described. Because this implementation is very close to the Kernighan and Ritchie standard, many of the sections apply exactly as written; these sections will not be commented upon. Any section not listed here can be assumed to be fully valid for the language accepted by the Lattice C compiler.

CRM 2.1 Comments

Although the default mode is that comments do not nest, a compile time option can be used to allow comments to be nested, so that each /* encountered must be matched by a corresponding */ before the comment terminates.

CRM 2.3 Keywords

Two additional keywords, void and enum, are reserved.

CRM 2.4.1 Integer constants

In addition to the standard integer constants, names declared as values for an enumeration type may be used as int constants.

CRM 2.4.3 Character constants

An additional two escape sequences are recognized: \v specifies a vertical tab (VT) character, and \x is used to introduce one or two hexadecimal digits which define the value of a single character. Thus,

```
'\xf9'
```

generates a character with the value ØxF9. Although by default the compiler permits only a single character to be defined, a compile time option can be used to permit multiple character constants. The result may be int or long, depending on the number of characters, and its value is machine-dependent.

CRM 2.5 Strings

The same \x convention described above can be employed in strings, where it is generally more useful. In addition, a compile time option can be used to force the compiler to recognize identically written string constants and only generate one copy of the string. (Note that strings used to initialize char arrays -- not char * -- are not actually generated, because they are really just shorthand for a comma-separated list of single-character constants.)

CRM 2.6 Hardware characteristics

See the implementation section of this manual for hardware characteristics.

CRM 4. WHAT'S IN A NAME?

Each enumeration is conceptually a separate type with its own set of named values. The properties of an enum type are identical to those of int type.

The void type is used as the type returned by functions that generate no return value; it thus represents the absence of any value.

CRM 6. CONVERSIONS

An expression can be converted to type void by means of a cast; the most common use of this is to make explicit the discarding of a return value from a function. An expresion of type void, however, cannot itself be converted or used in any way.

CRM 7.1 Primary expressions

The Lattice compiler always enforces the rules for the use of structures and unions for the simple reason that it cannot otherwise determine which list of member names is intended. Since the compiler maintains a separate list of members for each type of structure or union, the primary expression preceding the . or -> operator must be immediately recognizable as a structure or pointer to a structure of a specific type.

CRM 7.2 Unary operators

The requirement that the & operator can only be applied to an

lvalue is relaxed slightly to allow application to an array name (which is not considered an lvalue). Note that the meaning of such a construct is a pointer to the array itself, which is quite different from a pointer to the first element of the array. The difference between a pointer to an array and to an array's first element is only important when the pointer is used in an expression with an int offset, because the offset must be scaled (multiplied) by the size of the object to which the pointer points. In this case the target object size is the size of the whole array, rather than the size of a single element, if the pointer points to the array as a whole.

CRM 7.7 Equality operators

The only integer to which a pointer may be compared is the integer constant zero.

CRM 7.14 Assignment operators

Both operands of the simple assignment operator = may be structures or unions of the same type.

CRM 8.1 Storage class-specifiers

The text states that the storage class-specifier, if omitted from a declaration outside a function, is taken to be extern. This is somewhat misleading, if not plainly inaccurate; in fact (as the text points out in CRM 11.2), the presence or absence of extern is critical to determining whether an object is being defined or referenced. As noted in Section 2.2.4 of this document, if extern is present, then the declared object either exists in some other file or is defined later in the same file; if no storage class specifier is present, then the declared object is being defined and will be visible in other files. If the static specifier is present, the object is also defined but is not made externally visible. The only exception to these rules occurs for functions, where it is the presence of a defining statement body that determines whether the function is being defined.

The Lattice compiler can be forced to assume extern for all declarations outside a function by means of the -x compile time option. Declarations which explicitly specify static or extern are not affected.

CRM 8.2 Type specifiers

Two additional type specifiers are supported:

void
enum-specifier

The following additional multiple keyword types are recognized:

```
unsigned short
unsigned short int
unsigned long
unsigned long int
```

CRM 8.5 Structure and union declarations

The Lattice compiler treats the names of structure members quite differently from Kernighan and Ritchie. The names of members and tags do not conflict with each other or with the identifiers used for ordinary variables. Both structure and union tags are in the same class of names, so that the same tag cannot be used for both a structure and a union. A separate list of members is maintained for each structure; thus, a member name may not appear twice in a particular structure, but the same name may be used in several different structures within the same scope.

Enumerations are declared in much the same way as structures and unions, with the list of member declarations replaced by a simple list of identifiers. Enumerations are unique types which can only assume values from a list of named constants. The language treats them as int values but restricts operations on them to assignment and comparison. (The named constants, however, may appear wherever an int is legal.) The optional name which may follow the keyword enum plays the same role as the structure or union tag; it names a particular enumeration. All such names share the same space as structure and union tags. The names of enumerators in the same scope must be distinct from each other and from those of ordinary variables.

CRM 8.7 Type names

Although a structure or union may appear in a type name specifier, it must refer to an already known tag, that is, structure definitions cannot be made inside a type name. Thus, the sequence

```
(struct { int high, low; } *) x
```

is not permitted, but

```
struct HL { int high, low; };
. . .
(struct HL *) x
```

is acceptable.

CRM 10.2 External data definitions

The Lattice compiler applies a simple rule to external data declarations: if the keyword extern is present, the actual

storage will be allocated elsewhere, and the declaration is simply a reference to it. Otherwise, it is interpreted as an actual definition which allocates storage (unless the -x option has been used; see the comments on CRM 8.1).

CRM 12.3 Conditional compilation

The constant expression following #if may not contain the sizeof operator, and must appear on a single input line.

CRM 12.4 Line control

Although the filename for #line is denoted as identifier, it need not conform to the characteristics of C identifiers. The compiler takes whatever string of characters is supplied; the only lexical requirement for the filename is that it cannot contain any white space.

CRM 14.1 Structures and unions

Structures and unions may be assigned, passed as arguments to functions, and returned by functions.

The escape from typing rules described in the text is explicitly not allowed by the Lattice compiler. In a reference to a structure or union member, the name on the right must be a member of the aggregate named or pointed to by the expression of the left. This implementation, however, does not attempt to enforce any restrictions on reference to union members, such as requiring a value to be assigned to a particular member before allowing it to be examined via that member.

2.4 New Language Features

This section provides simple examples of the use of several of the new language features not described in the Kernighan and Ritchie text.

2.4.1 Void

The reserved word void is used to describe a function that has no return value:

```
void noval();              /* function with no return value */
void (*vp)();              /* pointer to void function */
```

While void operands may never be used in an expression, it is occasionally useful to cast something to void:

```
int xfunc();               /* function returning int */

(void)(xfunc(a, b));       /* discard return value */
```

The use of the cast shows that the programmer is intentionally discarding the return value from **xfunc**.

2.4.2 Enumerations

An enumerated data object is integral but may only assume values from a specified list of identifiers, which can be viewed as integer constants. The actual values assigned to the identifiers normally begin at zero and are incremented by one for each successive identifier in the list. An explicit value, however, can be forced for any identifier by using an = assignment. Subsequent identifiers are assigned that new value plus one, and so forth. Here is an example:

```
/* defining an enumerated data type */

enum color { red, blue, green = 4, puce, lavender };

/* defining enumerated data objects and pointers */

enum color x, *px;

/* using enumerated data objects */

x = red;
*px = x;
if (x == lavender) px = &x;
```

In this example, the symbols associated with the enumerated data type color are assigned the following values:

```
0 => red
1 => blue
4 => green
5 => puce
6 => lavender
```

2.4.3 Aggregate Assignment

Structures or unions of identical type may be copied by assignment:

```
struct XYZ
     {
     int x;
     double y;
     long z;
     } x, y, *px;

x = y;              /* copies contents of y to x */
*px = x;            /* copies x to the pointed to struct */
```

For purposes of assignment or passing to functions (see below), structures of identical type may also appear in a conditional expression:

```
y = (i > 30) ? x : *px;
```

2.4.4 Passing Aggregates by Value

A structure or union which appears in an argument list without a preceding ampersand (&) is passed by value to the function:

```
struct XYZ q;

functn(q);
```

The called function must make an appropriate declaration:

```
void functn(a);
struct XYZ a;
```

Because many existing programs pass the address of a structure without using an ampersand (&) operator, the compiler generates a warning whenever an aggregate is passed by value to a function. Note that the Lattice implementation of aggregate passing by value actually supplies the called function with a pointer which it uses to make a local copy of the caller's structure immediately upon entry.

2.4.5 Functions Returning Aggregates

A function may return an entire structure or union as a return value:

```
struct XYZ fxyz(a, b, c)
int a;
double b;
long c;
{
struct XYZ r;

r.x = a;
r.y = b;
r.z = c;
return (r);
}
```

The return value must be copied by assignment to another aggregate of the same type:

```
extern struct XYZ x, fxyz();
```

```
x = fxyz(2, 20., 2000L);
```

Note that the Lattice implementation actually returns a pointer to a static copy of the returned aggregate. Because this copy persists only long enough to assign the return value, such functions are still recursively reentrant (but not, in general, multi-tasking reentrant).

2.4.6 Function Declarations with Argument Types

External functions may be declared with an enclosed list of type names corresponding to expected arguments. When the function is called, the compiler checks arguments against the types specified in the prototype. Warning messages are issued if the actual argument expressions supplied to the function do not agree with the expected type or number of arguments. If additional arguments may be present for which no type checking is desired, the list of type names may be ended with a comma. Some examples are the following:

```
extern char *sbrk(int);
extern FILE *fopen(char *, char *);
extern double sin(double);
extern void fprintf(FILE *, char *, );
```

Three different warnings may be generated when calls to a function declared with argument types is made. If the number of arguments does not agree with those present in the prototype, warning 87 is issued. If an expression defining an argument does not agree in type with its corresponding declared type name, warning 88 is issued but the expression is not converted to the indicated type. If the expression is a constant, however, warning 89 is issued and the value is converted accordingly.

Section 3:
Standard Library Functions

In order to provide real portability, a C programming environment must provide -- in a machine-independent way -- not only a well-defined language but a library of useful functions as well. The portable library provided with the Lattice C compiler attempts to fulfill this requirement. Although not all of the features of these functions can be implemented on every system supported by the compiler, all systems must be able to provide the basic functions of memory allocation, file input/output, and character string manipulation; otherwise, the compiler itself could not be implemented. An important side benefit of presenting the functions from a machine-independent viewpoint is that it helps the programmer think of them as such.

When referring to the function descriptions presented in this section, remember that the compiler assumes that a function will return an int value unless it is explicitly declared otherwise. Any function which returns any other kind of value must be declared as that kind of function in advance of its first usage in the same file.

3.1 Memory Allocation Functions

The standard library provides memory allocation capabilities at several different levels. The higher level functions call the lower levels to perform the work, but provide easier interfaces in exchange for the extra overhead. The actual amount of memory available is system-dependent and usually depends on the size of the program.

All of the memory allocation functions return a pointer which is of type char *, but is guaranteed to be properly aligned to store any object.

3.1.1 Level 3 Memory Allocation

The functions described in this section provide a UNIX-compatible memory-allocation facility. The blocks of memory obtained may be released in any order, but it is an error to release something not obtained by calling one of these functions. Because these functions use overhead locations to keep track of allocation sizes, the free function does not require a size argument. The overhead does, however, decrease the efficiency with which these functions use the available memory. If many small allocations are requested, the available memory will be more efficently utilized if the level 2 functions are used instead.

NAME

malloc -- UNIX-compatible memory allocation

SYNOPSIS

```
p = malloc(nbytes);
char *p;              block pointer
unsigned nbytes;      number of bytes requested
```

DESCRIPTION

Allocates a block of memory in a way that is compatible with UNIX. The primary difference between malloc and getmem is that the former allocates a structure at the front of each block. This can result in very inefficient use of memory when making many small allocation requests.

RETURNS

p = NULL if not enough space available
  = pointer to block of nbytes of memory otherwise

CAUTIONS

Return value must be checked for NULL. The function should be declared char * and a cast operator used if defining a pointer to some other kind of object, as in:

```
char *malloc();
int *pi;
. . .
pi = (int *)malloc(N);
```

NAME

calloc -- allocate memory and clear

SYNOPSIS

```
p = calloc(nelt, eltsiz);
char *p;            block pointer
unsigned nelt;      number of elements
unsigned eltsiz;    element size in bytes
```

DESCRIPTION

Allocates and clears (sets to all zeros) a block of memory. The size of the block is specified by the product of the two parameters; this calling technique is obviously convenient for allocating arrays. Typically, the second argument is a sizeof expression.

RETURNS

p = NULL if not enough space available
  = pointer to block of memory otherwise

CAUTIONS

Return value must be checked for NULL. The function should be declared char * and a cast used if defining a pointer to some other kind of object, as in:

```
char *calloc();
struct buffer *pb;
. . .
pb = (struct buffer *)calloc(4, sizeof(struct buffer));
```

NAME

free -- UNIX-compatible memory release function

SYNOPSIS

```
ret = free(cp);
int ret;            return code
char *cp;           block pointer
```

DESCRIPTION

Releases a block of memory that was previously allocated by malloc or calloc. The pointer should be char * and is checked for validity, that is, verified to be an element of the memory pool.

RETURNS

```
ret =  Ø if successful
    = -1 if invalid block pointer
```

CAUTIONS

Remember to cast the pointer back to char *, as in:

```
char *malloc();
int *pi;
. . .
pi = (int *) malloc(N);
. . .
if (free((char *)pi) != Ø) { ... error ... }
```

3.1.2 Level 2 Memory Allocation

The functions described in this section provide an efficient and convenient memory allocation capability. Like the level 3 functions, allocation and de-allocation requests may be made in any order, and it is an error to free memory not obtained by means of one of these functions. The caller must retain both the pointer and the size of the block for use when it is freed; failure to provide the correct length may lead to wasted memory (the functions can detect an incorrect length when it is too large, but not when it is too small).

NAME

getmem/getml -- get a memory block

SYNOPSIS

```
p = getmem(nbytes);
p = getml(lnbytes);
char *p;            block pointer
unsigned nbytes;    number of bytes requested
long lnbytes;       long number of bytes requested
```

DESCRIPTION

These functions get a block of memory from the free memory pool. If the pool is empty or a block of the requested size is not available, more memory is obtained via the level 1 function sbrk. getml is provided only on those systems that offer more storage than an unsigned int can represent; it is otherwise identical to getmem.

RETURNS

p = NULL if not enough space available
  = pointer to memory block otherwise

CAUTIONS

Return value must be checked for NULL. The function should be declared char * and a cast used if defining a pointer to some other kind of object, as in:

```
char *getmem();
struct XYZ *px;
. . .
px = (struct XYZ *)getmem(sizeof(struct XYZ));
```

NAME

rlsmem/rlsml -- release a memory block

SYNOPSIS

```
ret = rlsmem(cp, nbytes);
ret = rlsml(cp, lnbytes);
int ret;                return code
char *cp;              block pointer to be freed
unsigned nbytes;      size of block
long lnbytes;         size of block as long integer
```

DESCRIPTION

These functions release the memory block by placing it on a free block list. If the new block is adjacent to a block on the list, they are combined. **rlsml** is provided only on those systems which offer more storage than an **unsigned int** can represent; it is otherwise identical to **rlsmem**.

RETURNS

ret = 0 if successful
= -1 if supplied block is not obtained by **getmem** or getml or if it overlaps one of the blocks on the list

CAUTIONS

Return value should be checked for error. If the correct size is not supplied, the block may not be freed properly.

3.1.3 Level 1 Memory Allocation

The two functions defined at the lowest level of memory allocation are primitives which perform the basic operations needed to implement a more sophisticated facility; they are used by the level 2 functions for that purpose. **sbrk** treats the total amount of memory available as a single block, from which portions of a specific size may be allocated at the low end, creating a new block of smaller size. **rbrk** merely resets the block back to its original size. The "break point" mentioned here should not be confused with the breakpoint concept used in debugging; this term simply refers to the address of the low end of the block of memory manipulated by **sbrk**.

NAME

sbrk/lsbrk -- set memory break point

SYNOPSIS

```
p = sbrk(nbytes);
p = lsbrk(lnbytes);
char *p;            points to low allocated address
unsigned nbytes;    number of bytes to be allocated
long lnbytes;       long number of bytes to be allocated
```

DESCRIPTION

These functions allocate a block of memory of the requested size, if possible; they form the basic UNIX memory allocator. Memory is allocated by advancing the "memory break point," which is simply the base address of a block of memory whose location is system-dependent. The previous break point address is then returned to the caller. **lsbrk** is provided only on those systems which offer more storage than an **unsigned int** can represent; except for the NULL return code, it is otherwise identical to **sbrk**.

RETURNS

```
p = -1 if request cannot be fulfilled (sbrk only)
p =  Ø if request cannot be fulfilled (lsbrk only)
  =  pointer to low address of block if successful
```

CAUTIONS

For consistency with the UNIX function, **sbrk** returns -1 if it cannot satisfy the request, although the rest of the memory allocators return NULL. Both functions should be declared **char *** and a cast used if defining a pointer to some other kind of object.

NAME

rbrk -- reset memory break point

SYNOPSIS

```
rbrk();
```

DESCRIPTION

Resets the memory break point to its original starting-point. This effectively frees all memory allocated by any of the memory allocation functions.

CAUTIONS

Like **rstmem** above, this function cannot be used if any files are open and being accessed using the level 2 I/O functions.

3.2 I/O and System Functions

The standard library provides I/O functions at several different levels, with single character get and put functions and formatted I/O at the highest levels, and direct byte stream I/O functions at the lowest levels.

Two general classes of I/O functions are provided. First, the level 2 functions define a buffered text file interface which implements the single character I/O functions as macros rather than function calls. Second, the level 1 functions define a byte stream-oriented file interface, primarily useful for manipulation of disk files, though most of the same functions are applicable to devices (such as the user's console) as well.

In general, these functions adhere to the UNIX convention for reporting errors. When a failure indication from an I/O function is obtained, programmers can inspect the global integer errno, which will contain one of the error codes defined in the header file **error.h**. Additional information may be available from the global integer _oserr, which contains an operating system error code, if applicable. Check the implementation section of the manual for details.

The system functions discussed in this section are concerned with program termination and transfer of control. Additional system functions are described in the implementation section of the manual.

3.2.1 Level 2 I/O Functions and Macros

These functions provide a buffered interface using a special structure, manipulated internally by the functions, to which a pointer called the file pointer is defined. This structure is defined in the standard I/O header file (called **stdio.h** on most systems) which generally must be included (by means of a **#include** statement) in the source file where level 2 features are being used. The file pointer is used to specify the file upon which operations are to be performed. Some functions require a file pointer, such as

```
FILE *fp;
```

to be explicitly included in the calling sequence; others imply a specific file pointer. In particular, the file pointers **stdin** and **stdout** are implied by the use of several functions and macros; these files are so commonly used that on most systems they are opened automatically before the **main** function of a program begins execution. Other file pointers must be declared by the programmer and initialized by calls to the **fopen** function.

The level 2 functions are designed to work primarily with text

files. The usual C convention for line termination uses a single character, the newline (\n), to indicate the end of a line.

The actual I/O operations are performed by the level 2 functions through calls to the level 1 I/O functions described in the next section. The normal mode of buffering, designed to support sequential operations, performs read and write functions in 512-byte blocks.

Normally the level 2 functions acquire buffers via the level 2 memory allocator unless the file is on a device other than a disk. Alternatively, the setbuf function allows a private buffer to be attached. This function assumes that the buffer is the standard size, which is defined via the BUFSIZ constant in stdio.h. If for some reason operating the level 2 I/O functions in the buffered mode is not desirable, the setnbf function can be called. This is done automatically for non-disk files or if setbuf is called with a NULL buffer pointer.

In the descriptions below, some of the function calls are actually implemented as macros; these are noted explicitly. The reason the programmer should be aware of the distinction is because many macros involve the conditional operator and may, under certain conditions, evaluate an argument expression more than once. This can cause unexpected results if that expression involves side effects, such as increment or decrement operators or function calls. In addition, unlike functions, macros do not have addresses, making it impossible for pointers to them to be passed to other functions.

NAME

fopen/freopen -- open/re-open a buffered file

SYNOPSIS

```
fp = fopen(name, mode);
fp = freopen(name, mode, fpx);
FILE *fp;          file pointer for specified file
char *name;        file name
char *mode;        access mode
FILE *fpx;         existing file pointer (freopen only)
```

DESCRIPTION

These functions open a file for buffered access. fopen returns a file pointer after finding a free slot in a predefined array (_iob), while freopen uses the file pointer supplied by the caller (useful for opening stdin, etc.); note that this file pointer is automatically closed by freopen before being reused. On most systems, no more than 20 files (including stdin, stdout, and stderr, if those are opened for main) can be opened using fopen or freopen. The null-terminated string which specifies the filename must conform to local file naming conventions. The access mode is also specified as a string, and may be one of the following:

r open for reading (mode set according to _fmode)

w open for writing (mode set according to _fmode)

a open for appending (mode set according to _fmode)

The mode characters must be specified in lower case. The a option adds to the end of an existing file, or creates a new one; the w option discards any data in the file, if it already exists; the r option simply reads an existing file. Opening the file to append forces all data to be written to the current end of file, regardless of previous seeks.

Any of the above strings may be appended with a plus sign + to indicate opening for update (both reading and writing). In this mode, both reads and writes may be performed on the file; in order to switch between reading and writing, however, an fseek or rewind must be executed. If a file is opened for reading with a plus, then the file must already exist; but if a file is opened for writing with a plus, the file will be created anew. Opening for appending with a plus will permit reads to take place from any position in the file, but all write operations will occur at the end of the file.

RETURNS

fp = NULL if error
= file pointer for specified file if successful

CAUTIONS

The return code must be checked for NULL; the error return may be generated if an invalid mode was specified or if the file was not found, could not be created, or too many files were already open.

NAME

fclose -- close a buffered file

SYNOPSIS

```
ret = fclose(fp);
int ret;            return code
FILE *fp;           file pointer for file to be closed
```

DESCRIPTION

Completes the processing of a file and releases all related resources. If the file was being written, any data which has accumulated in the buffer is written to the file, and the level 1 close function is called for the associated file descriptor. The buffer associated with the file block is freed. fclose is automatically called for all open files when a program calls the exit function (see Section 3.2.3) or when the main program returns, but it is good programming practice to close files explicitly. As the last buffer is not written until fclose is called, data may be lost if an output file is not properly closed.

RETURNS

ret = -1 if error
= 0 if successful

NAME

getc/getchar -- get character from file

SYNOPSIS

```
c = getc(fp);
c = getchar();
int c;              next input character or EOF
FILE *fp;           file pointer
```

DESCRIPTION

Gets the next character from the indicated file (stdin, in the case of getchar). The value EOF (-1) is returned on end-of-file or error.

RETURNS

c = character
  = EOF if end-of-file or error

CAUTIONS

Implemented as macros, so beware of side effects.

NAME

putc/putchar -- put character to file

SYNOPSIS

```
r = putc(c, fp);
r = putchar(c);
int r;              same as character sent, or error code
char c;             character to be output
FILE *fp;           file pointer
```

DESCRIPTION

Puts the character to the indicated file (stdout, in the case of putchar). The value EOF (-1) is returned on end-of-file or error.

RETURNS

r = character sent if successful
  = EOF if error or end-of-file

CAUTIONS

Implemented as macros, so beware of side effects.

NAME

fgetc/fputc -- get/put a character

SYNOPSIS

```
r = fgetc(fp);
r = fputc(c, fp);
int r;              return character or code
char c;             character to be sent (fputc)
FILE *fp;           file pointer
```

DESCRIPTION

These functions get (fgetc) or put (fputc) a single character to the indicated file. Since they are functions, they are often recommended for use rather than the corresponding macros (getc and putc) in two types of situations: (1) if many calls are made and/or (2) if the programmer is concerned about the amount of memory used in the macro expansions. The tradeoff is the usual one: the macro executes more quickly because it saves a function call; the function requires less memory since its code is present in the program only once.

RETURNS

r = character if successful (c, for fputc)
  = EOF if error or end-of-file

NAME

ungetc -- push character back on input file

SYNOPSIS

```
r = ungetc(c, fp);
int r;              return character or code
char c;             character to be pushed back
FILE *fp;           file pointer
```

DESCRIPTION

Pushes back a character to the specified input file. The character supplied must be the character most recently obtained by a getc (or getchar, in which case fp should be supplied as stdin) invocation.

RETURNS

r = character if successful
  = EOF if previous character does not match

NAME

fread/fwrite -- read/write blocks of data from/to a file

SYNOPSIS

```
nact = fread(p, s, n, fp);
nact = fwrite(p, s, n, fp);
int nact;          actual number of blocks read or written
char *p;           pointer to first block of data
int s;             size of each block, in bytes
int n;             number of blocks to be read or written
FILE *fp;          file pointer
```

DESCRIPTION

These functions read (fread) or write (fwrite) blocks of data from or to the specified file. Each block is of size s bytes; blocks start at p and are stored contiguously from that location. n specifies the number of blocks (of size s) that are to be read or written.

RETURNS

nact = actual number of blocks (of size s) read or written; may be less than n if error or end-of-file occurred

CAUTIONS

Return value must be checked to verify that the correct number of blocks was processed. The ferror and feof macros can be used to determine the cause if the return value is less than n.

NAME

gets/fgets -- get a string

SYNOPSIS

```
p = gets(s);
p = fgets(s, n, fp);
char *p;            returned string pointer
char *s;            buffer for input string
int n;              number of bytes in buffer
FILE *fp;           file pointer
```

DESCRIPTION

Gets an input string from a file. The specified file (stdin, in the case of gets) is read until a newline is encountered or n-1 characters have been read (fgets only). Then, gets replaces the newline with a null byte, while fgets passes the newline through with a null byte appended.

RETURNS

```
p = NULL if end of file or error
  = s if successful
```

CAUTIONS

For gets, there is no length parameter; thus, the input buffer provided must be large enough to accommodate the string.

NAME

puts/fputs -- put a string

SYNOPSIS

```
r = puts(s);
r = fputs(s, fp);
int r;              return code
char *s;            output string pointer
FILE *fp;           file pointer
```

DESCRIPTION

Puts an output string to a file. Characters from the string are written to the specified file (stdout, in the case of puts) until a null byte is encountered. The null byte is not written, but puts appends a newline.

RETURNS

r = EOF if end-of-file or error

NAME

scanf/fscanf/sscanf -- perform formatted input conversions

SYNOPSIS

```
n = scanf(cs, ...ptrs...);
n = fscanf(fp, cs, ...ptrs...);
n = sscanf(ss, cs, ...ptrs...);
int n;              number of input items matched, or EOF
FILE *fp;           file pointer (fscanf only)
char *ss;           input string (sscanf only)
char *cs;           format control string
---- ...ptrs...;    pointers for return of input values
```

DESCRIPTION

These functions perform formatted input conversions on text obtained from three types of files:

(1) the stdin file (scanf);
(2) the specified file (fscanf); or
(3) the specified string (sscanf).

The control string contains format specifiers and/or characters to be matched from the input; the list of pointer arguments specify where the results of the conversions are to go. Format specifiers are of the form

%[*][n][l]X

where

(1) the optional * means that the conversion is to be performed, but the result value not returned;
(2) the optional **n** is a decimal number specifying a maximum field width;
(3) the optional **l** (el) is used to indicate a long int or long float (i.e., double) result is desired;
(4) **X** is one of the format type indicators from the following list:

d -- decimal integer
o -- octal integer
x -- hexadecimal integer
h -- short integer
c -- single character
s -- character string
f -- floating point number

The format type must be specified in lower case. White space characters in the control string are ignored;

characters other than format specifiers are expected to match the next non-white space characters in the input. The input is scanned through white space to locate the next input item in all cases except the c specifier, where the next input character is returned without this initial scan. Note that the %s specifier terminates on any white space. See the Kernighan and Ritchie text for a more detailed explanation of the formatted input functions.

RETURNS

n = number of input items successfully matched, i.e., for which valid text data was found; this includes all single character items in the control string
= EOF if end-of-file or error is encountered during scan

CAUTIONS

All of the input values must be pointers to the result locations. Make sure that the format specifiers match up properly with the result locations. If the assignment suppression feature (*) is used, remember that a pointer must not be supplied for that specifier.

NAME

printf/fprintf/sprintf -- generate formatted output

SYNOPSIS

```
printf(cs, ...args...);
fprintf(fp, cs, ...args...);
n = sprintf(ds, cs, ...args...);
int n;              number of characters (sprintf only)
FILE *fp;           file pointer (fprintf)
char *ds;           destination string pointer (sprintf)
char *cs;           format control string
---- ...args...;    list of arguments to be formatted
```

DESCRIPTION

These functions perform formatted output conversions and send the resulting text to:

(1) the stdout file (printf);
(2) the specified file (fprintf); or
(3) the specified output string (sprintf).

The control string contains ordinary characters, which are sent without modification to the appropriate output, and format specifiers of the form

%[-][m][.p][l]**X**

where

(1) the optional - indicates the field is to be left justified (right justified is the default);
(2) the optional **m** field is a decimal number specifying a minimum field width;
(3) the optional **.p** field is the character . followed by a decimal number specifying the precision of a floating point image or the maximum number of characters to be printed from a string;
(4) the optional **l** (el) indicates that the item to be formatted is long; and
(5) **X** is one of the format type indicators from the following list:

d -- decimal signed integer
u -- decimal unsigned integer
x -- hexadecimal integer
o -- octal integer
s -- character string
c -- single character
f -- fixed decimal floating point

e -- exponential floating point
g -- use e or f format, whichever is shorter

The format type must be specified in lower case. Characters in the control string which are not part of a format specifier are sent to the appropriate output; a % may be sent by using the sequence %%. See the Kernighan and Ritchie text for a more detailed explanation of the formatted output functions.

RETURNS

n = number of characters placed in ds (sprintf only), not including the null byte terminator

CAUTIONS

For sprintf, no check of the size of the output string area is made; thus, the buffer provided must be large enough to contain the resulting image. In all cases, the format specifiers must match up properly with the supplied values for formatting.

NAME

fseek -- seek to a new file position

SYNOPSIS

```
ret = fseek(fp, pos, mode);
int ret;            return code
FILE *fp;           file pointer
long pos;           desired file position
int mode;           offset mode
```

DESCRIPTION

Seeks to a new position in the specified file. See the lseek function description (Section 3.2.2) for the meaning of the offset mode argument.

RETURNS

```
ret =  0 if successful
    = -1 if error
```

NAME

ftell -- return current file position

SYNOPSIS

```
pos = ftell(fp);
long pos;           current file position
FILE *fp;           file pointer
```

DESCRIPTION

Returns the current file position, that is, the number of bytes from the beginning of the file to the byte at which the next read or write operation will transfer data.

RETURNS

pos = current file position (long)

CAUTIONS

The file position returned takes account of the buffering used on the file, so that the file position returned is a logical file position rather than the actual position.

NAME

ferror/feof -- check if error/end of file

SYNOPSIS

```
ret = feof(fp);
ret = ferror(fp);
int ret;            return code
FILE *fp;           file pointer
```

DESCRIPTION

These macros generate a non-zero value if the indicated condition is true for the specified file.

RETURNS

ret = non-zero if error (ferror) or end of file (feof)
= zero if not

NAME

clrerr/clearerr -- clear error flag for file

SYNOPSIS

```
clrerr(fp);
clearerr(fp);
FILE *fp;           file pointer
```

DESCRIPTION

Clears the error flag for the specified file. Once set, the flag will remain set, forcing EOF returns for functions on the file, until this function is called.

NAME

fileno -- return file number for file pointer

SYNOPSIS

```
fn = fileno(fp);
int fn;         file number associated with file pointer
FILE *fp;       file pointer
```

DESCRIPTION

Returns the file number, used for the level 1 I/O calls, for the specified file pointer.

RETURNS

fn = file number (file descriptor) for level 1 calls

CAUTIONS

Implemented as a macro.

NAME

rewind -- rewind a file

SYNOPSIS

```
rewind(fp);
FILE *fp;           file pointer
```

DESCRIPTION

Resets the file position of the specified file to the beginning of the file.

CAUTIONS

Implemented as a macro.

NAME

fflush -- flush output buffer for file

SYNOPSIS

```
fflush(fp);
FILE *fp;           file pointer
```

DESCRIPTION

Flushes the output buffer of the specified file, that is, forces it to be written.

CAUTIONS

This macro must be used only on files which have been opened for writing or appending.

NAME

setbuf -- change buffer for level 2 file I/O

SYNOPSIS

```
setbuf(fd,buf);
FILE *fp;           file pointer for file
char *buf;          pointer to buffer to be attached
```

DESCRIPTION

Attaches a private buffer to the file whose file pointer is fp. The length of the buffer is assumed to be the same as _bufsiz, which is defaulted to the constant BUFSIZ defined in stdio.h.

If the buffer pointer is NULL, then this function is equivalent to setnbf.

CAUTIONS

buf must be large enough to accommodate _bufsiz charcacters.

NAME

setnbf -- set file unbuffered

SYNOPSIS

```
setnbf(fp);
FILE *fp;           file pointer
```

DESCRIPTION

Changes the buffering mode for the specified file pointer from the default 512-byte block mode to the unbuffered mode used for devices (including the user's console). In this mode, read and write operations are performed using single characters.

CAUTIONS

Although the unbuffered mode may be used without difficulty on files, the standard buffering mode is generally more efficient. Thus, this function should only be used for those "files" which are definitely known to be devices.

3.2.2 Level 1 I/O Functions

These functions provide a basic, low-level I/O interface which allows a file to be viewed as a stream of randomly addressable bytes. Operations are performed on the file using the functions described in this section; the file is specified by a file number or file descriptor, such as

```
int fd;
```

which is returned by open or creat when the file is opened. Data may be read or written in blocks of any size, from a single byte to as much as several kilobytes in a single operation. The concept of a file position is key: the file position is a long integer, such as

```
long fpos;
```

which specifies the position of a byte in the file as the number of bytes from the beginning of the file to that particular byte. Thus, the first byte in the file is at file position ØL. Two distinct file positions are maintained internally by the level 1 functions. The current file position is the point at which data transfers take place between the program and the file; it is set to zero when the file is opened, and is advanced by the number of bytes read or written using the read and write functions. The end of file position is simply the total number of bytes contained in the file; it is changed only by write operations which increase the size of the file.

The current file position can be set to any value from zero up to and including the end of file position using the lseek function. Thus, to append data to a file, the current file position is set to the end of the file using lseek before any write operations are performed. When data is read from near the end of file, as much of the requested count as can be satisfied is returned; zero is returned for attempts to read when the file position is at the end of file.

Although the level 1 functions are primarily useful for working with files, they can be used to read and write data to devices (including the user's terminal), as well. The exact nature of the I/O performed is system-dependent, but it is generally unbuffered. The lseek function has no effect on devices, and usually returns an error status.

The actual I/O operations on disk files are buffered, but at a level that is generally transparent to the programmer. The buffering makes close operations a necessity for files that are modified.

NAME

open -- open a file

SYNOPSIS

```
file = open(name, rwmode);
int file;          file number or error code
char *name;        file name
int  mode;         indicates read/write mode and other
                   options (see below)
```

DESCRIPTION

Opens a file for access using the level 1 I/O functions. The file name must conform to local naming conventions. The mode word indicates the type of I/O which will be performed on the file. The header file fcntl.h defines the codes for the mode arguments:

O_RDONLY	Read only access
O_WRONLY	Write only access
O_RDWR	Read/write access

Also, the following flags can be ORed into the above codes:

O_CREAT	Create the file if it doesn't exist
O_TRUNC	Truncate (set to zero length) the file if it does exist
O_EXCL	Force create to fail if file exists
O_APPEND	Seek to end-of-file before each write

The current file position is set to zero if the file is successfully opened. On most systems, no more than 20 files (including any which are being accessed through the level 2 functions, such as stdin, stdout, etc.) can be open at the same time. Closing the file releases the file number for use with some other file.

RETURNS

file = file number to access file, if successful
= -1 if error

CAUTIONS

Check the return value for error.

NAME

creat -- create a new file

SYNOPSIS

```
file = creat(name, pmode);
int file;           file number or error code
char *name;         file name
int pmode;          access privilege mode bits
```

DESCRIPTION

Creates a new file with the specified name and prepares it for access via the level 1 I/O functions. The file name must conform to local naming conventions. Creating a device is equivalent to opening it. The access privilege mode bits are system-dependent and on some systems may be largely ignored. If the file already exists, its contents are discar ded. The current file position and the end-of-file are both zero (indicating an empty file) if the function is successful.

RETURNS

file = file number to access file, if successful
= -1 if error

CAUTIONS

Check the return value for error. creat should be used only on files which are being completely rewritten, since any existing data is lost.

NAME

unlink/remove -- remove file name from file system

SYNOPSIS

```
ret = unlink(name);
ret = remove(name);
int ret;            return code: Ø if successful
char *name;         name of file to be removed
```

DESCRIPTION

Removes the specified file from the file system. The file name must conform to local naming conventions. The specified file must not be currently open. All data in the file is lost.

RETURNS

```
ret = Ø if successful
    = -1 if error
```

CAUTIONS

Should be used with care since the file, once removed, is generally irretrievable.

NAME

rename -- rename a file

SYNOPSIS

```
error = rename(old,new);
int error;              Ø for success
char *old;              old file name
char *new;              new file name
```

DESCRIPTION

This function renames a file, if possible. A failure will occur if the new file name already exists or if the old file name does not. The specified file must not be currently open.

RETURNS

error = Ø if file successfully renamed
= -1 if error

NAME

read -- read data from file

SYNOPSIS

```
status = read(file, buffer, length);
int status;          status code or actual length
int file;            file number for file
char *buffer;        input buffer
int length;          number of bytes requested
```

DESCRIPTION

Reads the next set of bytes from a file. The return count is always equal to the number of bytes placed in the buffer and will never exceed the length parameter, except in the case of an error, where -1 is returned. The file position is advanced accordingly.

RETURNS

```
status = Ø if end-of-file
       = -1 if error occurred
       = number of bytes actually read, otherwise
```

CAUTIONS

If fewer than the requested number of bytes remain between the current file position and the end-of-file, only that number is transferred and returned.

NAME

write -- write data to file

SYNOPSIS

```
status = write(file, buffer, length);
int status;             status code or actual length
int file;               file number
char *buffer;           output buffer
int length;             number of bytes in buffer
```

DESCRIPTION

Writes the next set of bytes to a file. The return count is equal to the number of bytes written, unless an error occurred. The file position is advanced accordingly.

RETURNS

```
status = -1 if error
       = number of bytes actually written
```

CAUTIONS

The number of bytes written may be less than the supplied count if a physical end-of-file limitation was encountered.

NAME

lseek -- seek to specified file position

SYNOPSIS

```
pos = lseek(file, offset, mode);
long pos;          returned file position or error code
int file;          file number for file
long offset;       desired position
int mode;          offset mode:
                   Ø = relative to beginning of file
                   1 = relative to current file position
                   2 = relative to end-of-file
```

DESCRIPTION

Changes the current file position to a new position in the file. The offset is specified as a long int and is added to the current position (mode 1) or to the end-of-file (mode 2).

RETURNS

pos = -1L if error occurred
= new file position if successful

CAUTIONS

The offset parameter must be a long quantity; therefore a long constant should be indicated when supplying a zero. In most cases, the return code should be checked for error, which indicates that an invalid file position (beyond the end-of-file) was specified. Note that the current file position may be obtained by

```
long cpos, lseek();
. . .
cpos = lseek(file, ØL, 1);
```

which will never return an error code.

NAME

close -- close a file

SYNOPSIS

```
status = close(file);
int status;              status code: Ø if successful
int file;                file number
```

DESCRIPTION

Closes a file and frees the file number for use in accessing another file. Any buffers allocated when the file was opened are released.

RETURNS

```
status = Ø if successful
       = -1 if error
```

CAUTIONS

This function must be called if the file was modified; otherwise, the end-of-file and the actual data on disk may not be updated properly.

3.2.3 Program Exit and Jump Functions

The program entry mechanism, that is, the means by which the main function gains control, is sufficiently system-dependent that it must be described in the implementation section of this manual. Program exit, however, is somewhat more general, although not without its own implementation dependencies.

The simplest way to terminate execution of a C program is for the main function to execute a **return** statement, or -- even simpler -- to "drop through" its terminating brace. In many cases, however, a more flexible program exit capability is needed; this is provided by the exit and _exit functions described in this section. They offer the advantage of allowing any function -- not just main -- to cause termination of the program, and in some systems, they allow information to be passed to other programs.

In some cases, it is useful for a program to be able to pass control directly to another part of the program (within a different function) without having to go through a long and possibly complicated series of function returns. The setjmp and longjmp functions provide a general capability for achieving this.

NAME

exit -- terminate execution of program and close files

SYNOPSIS

```
exit(errcode);
int errcode;        exit error code
```

DESCRIPTION

Terminates execution of the current program, but first closes all output files which are currently open through the level 2 I/O functions. The error code is normally set to zero to indicate no error, and to a non-zero value if some kind of error exit was taken.

NAME

_exit -- terminate execution immediately

SYNOPSIS

```
_exit(errcode);
int errcode;        exit error code
```

DESCRIPTION

Terminates execution of the current program immediately, without checking for open files.

NAME

setjmp/longjmp -- perform non-local goto

SYNOPSIS

```
ret = setjmp(save);
longjmp(save,value);

int ret;            return code
int value;          return value
jmp_buf save;       context save buffer
```

DESCRIPTION

The setjmp function saves the current stack mark in the buffer area specified by save and returns a value of Ø. A subsequent call to longjmp will then cause control to return to the next statement after the original setjmp call, with value as the return code. If value is Ø, it is forced to 1 by longjmp.

The jmp_buf descriptor is defined in the header file called setjmp.h.

This mechanism is useful for quickly popping back up through multiple layers of function calls under exceptional circumstances. Structured programming gurus lose a lot of sleep over the "pathological connections" that can result from indiscriminate usage of these functions.

CAUTIONS

Calling longjmp with an invalid save area is an effective way to disrupt the system. One common error is to use longjmp after the function calling setjmp has returned to its caller. This cannot hope to succeed, since the stack frame for that function is no longer present on the stack.

3.3 Utility Functions and Macros

The portable library provides a variety of additional functions useful for many of the common data manipulations performed by C programs. Three utilities provide fast memory transfers; a set of macros allows quick testing of character types; and several utility functions facilitate character string handling.

3.3.1 Memory Utilities

The three utility functions described here are usually implemented in machine language for maximum efficiency. These are the equivalent of the almost universal FILL and MOVE subroutines defined in many other languages.

NAME

setmem -- initialize memory to specified char value

SYNOPSIS

```
setmem(p, n, c);
char *p;          base of memory to be initialized
unsigned n;       number of bytes to be initialized
char c;           initialization value
```

DESCRIPTION

Sets the specified number of bytes of memory to the specified byte value. On many systems a hardware block fill instruction is used to perform the initialization. This function is useful for the initialization of auto char arrays.

CAUTIONS

Some systems may distinguish between char * pointers and pointers of other types, so that it is good practice to use a cast operator when arrays or pointers of other types are used for the p argument.

NAME

movmem -- move a block of memory

SYNOPSIS

```
movmem(s, d, n);
char *s;            source memory block
char *d;            destination memory block
unsigned n;         number of bytes to be transferred
```

DESCRIPTION

Moves memory from one location to another. The function checks the relative locations of source and destination blocks, and performs the move in the order necessary to preserve the data in the event of overlap. On many systems a hardware block move instruction is used to perform the transfer.

CAUTIONS

Some systems may distinguish between char * pointers and pointers of other types, so that it is good practice to use a cast operator when arrays or pointers of other types are used for the s and d arguments.

NAME

repmem -- replicate values through memory

SYNOPSIS

```
repmem(s, v, lv, nv);
char *s;            memory to be initialized
char *v;            template of values to be replicated
int lv;             number of bytes in template
int nv;             number of templates to be replicated
```

DESCRIPTION

Replicates a set of values throughout a block of memory. This function is a generalized version of **setmem**, and can be used to initialize arrays of items other than **char**. Note that the replication count indicates the number of copies of **v** which are to be made, not the total number of bytes to be initialized.

CAUTIONS

Some systems may distinguish between **char *** pointers and other types of pointers, so that it is good practice to use a cast operator when arrays or pointers of other types are used for the **s** and **v** arguments.

3.3.2 Character Type Macros and Functions

The character type header file, called ctype.h on most systems, defines several macros which are useful in the analysis of text data. Most allow the programmer to determine quickly the type of a character, i.e., whether it is alphabetic, numeric, punctuation, etc. These macros refer to an external array called _ctype which is indexed by the character itself, and so they are generally much faster than functions which check the character against a range or discrete list of values. Although ASCII is defined as a 7-bit code, the _ctype array is defined to be 257 bytes long so that valid results are obtained for any character value. This means that a character with the value Øxbl, for instance, will be classified the same as a character with the value Øx31. Programs that need to distinguish between these values must test for the Øx8Ø bit before using one of these macros. Note that _ctype is actually indexed by the character value plus one; this allows the standard EOF value (-1) to be tested in a macro without yielding a nonsense result. EOF yields a zero result for any of the macros: it is not defined as any of the character types.

The following list presents the macros defined in the character type header file ctype.h. Note that many of these will evaluate argument expressions more than once; beware of using expressions with side effects, such as function calls or increment or decrement operators. Note that the file ctype.h must be included if any of these macros are used; otherwise, the compiler will generate a reference to a function of the same name. Those macros marked with a '*' are also available in function form. In order to use the function form, do not #include the ctype.h header file in that source file. If some of the other capabilities of ctype.h are needed, the header file should be included anyway; #undef directives can be used for the specific macros that need to be treated as functions.

*	isalpha(c)	non-zero if c is alphabetic, Ø if not
*	isupper(c)	non-zero if c is upper case, Ø if not
*	islower(c)	non-zero if c is lower case, Ø if not
*	isdigit(c)	non-zero if c is a digit Ø-9, Ø if not
	isxdigit(c)	non-zero if c is a hexadecimal digit, Ø if not (Ø-9, A-F, a-f)
*	isspace(c)	non-zero if c is white space, Ø if not
	ispunct(c)	non-zero if c is punctuation, Ø if not
*	isalnum(c)	non-zero if c is alphabetic or digit
	isprint(c)	non-zero if c is printable (including blank)
	isgraph(c)	non-zero if c is graphic (excluding blank)
*	iscntrl(c)	non-zero if c is control character
	isascii(c)	non-zero if c is ASCII (Ø-127)
	iscsym(c)	non-zero if valid character for C

		identifier, Ø if not
	iscsymf(c)	non-zero if valid first character for C identifier, Ø if not
*	toupper(c)	converts c to upper case, if lower case
*	tolower(c)	converts c to lower case, if upper case
	toascii(c)	removes high bit from c

Note that the last two macros generate the value of c unchanged if it does not qualify for the conversion.

3.3.3 String Utility Functions

The portable library provides several functions to perform many of the most common string manipulations. These functions all work with sequences of characters terminated by a null (zero) byte, which is the C definition of a character string. A special naming convention is used, which works as follows: The first two characters of a string function are always st, while the third character indicates the type of the return value from the function:

stc	function returns an int count
stp	function returns a character pointer
sts	function returns an int status value

Thus, the name of the function shows at a glance the type of value it returns.

For compatibility with other C implementations, several other function whose names begin with str have also been provided.

NAME

strcat/strncat -- concatenate strings

SYNOPSIS

```
to = strcat(to, from);
to = strncat(to, from, max);

char *to;               destination string
char *from;             source string
int max;                maximum number of characters
```

DESCRIPTION

These functions append the "from" string to the "to" string. For strncat, no more than the specified maximum number of characters will be appended. The result is always null-terminated.

RETURNS

to = pointer to result (same as original to argument)

CAUTIONS

strncat should be used if there is any question that the destination string might not be large enough to hold the result. Either function must be declared char * if the return value is to be used. The "to" and "from" must not reference the same string.

NAME

strlen/stclen -- measure length of string

SYNOPSIS

```
length = stclen(s);
length = strlen(s);
int length;          number of bytes in s (before null)
```

DESCRIPTION

These functions count the number of bytes in s before the null terminator. The terminator itself is not included in the count. strlen is provided for compatibility with other implementations.

RETURNS

length = number of bytes in string before null byte

NAME

strcpy/strncpy/stpcpy/stccpy -- copy one string to another

SYNOPSIS

```
to = strcpy(to, from);
to = strncpy(to, from, length);
to = stpcpy(to, from);
actual = stccpy(to, from, length);
int actual;          actual number of characters moved
                     (stccpy only)
char *to;            destination string pointer
char *from;          source string pointer
int length;          maximum length of copy
```

DESCRIPTION

These functions move the null-terminated source string to the destination string. For strncpy and stccpy, if the source is too long for the destination, its rightmost characters are not moved. The destination string is always null-terminated.

RETURNS

to = pointer to destination string (same as original to argument) (strcpy, strncpy, stpcpy)
actual = actual number of characters moved, including the null terminator (stccpy only)

CAUTIONS

strncpy or stccpy should be used if there is any question that the destination string might not be large enough to hold the result. Functions returning char * must be so declared before being used.

NAME

strcmp/strncmp/stscmp -- compare two strings

SYNOPSIS

```
status = strcmp(s, t);
status = strncmp(s, t, length);
status = stscmp(s, t);
int status;              result of comparison
                         >Ø if s>t, Ø if s==t, <Ø if s<t
char *s;                 first string to compare
char *t;                 second string to compare
int length;              length of comparison (strncmp only)
```

DESCRIPTION

These functions compare two null-terminated strings, byte by byte, and return an int status indicating the result of the comparison. If zero, the strings are identical, up to and including the terminating byte. If non-zero, the status indicates the result of the comparison of the first pair of bytes which were not equal. For strncmp, no more than the specified number of characters will be compared.

RETURNS

```
status = Ø if strings match
       < Ø if first string less than second string
       > Ø if first string greater than second string
```

CAUTIONS

The result of the comparison may depend on whether characters are considered signed, if any of the characters is greater than 127.

NAME

stcu_d -- convert unsigned integer to decimal string

SYNOPSIS

```
length = stcu_d(out, in, outlen);
int length;          output string length (excluding null)
char *out;           output string
unsigned in;         input value
int outlen;          sizeof(out)
```

DESCRIPTION

Converts an unsigned integer into a string of decimal digits terminated with a null byte. Leading zeros are not copied to the output string, and if the input value is zero, only a single Ø character is produced.

RETURNS

length = number of characters placed in output string, not including the null terminator

CAUTIONS

If the output string is too small for the result, only the rightmost digits are returned. Note that outlen must be one larger than the largest number of digits.

NAME

stci_d -- convert signed integer to decimal string

SYNOPSIS

```
length = stci_d(out, in, outlen);
int length;          output string length (excluding null)
char *out;           output string
int in;              input value
int outlen;          sizeof(out)
```

DESCRIPTION

Converts an integer into a string of decimal digits terminated with a null byte. If the integer is negative, the output string is preceded by a -. Leading zeros are not copied to the output string.

RETURNS

length = number of characters placed in output string, not including the null terminator

CAUTIONS

If the output string is too small for the result, the returned length may be zero, or a partial string may be returned. Note that outlen must be two larger than the largest number of digits.

NAME

stch_i -- convert hexadecimal string to integer

SYNOPSIS

```
count = stch_i(p, r);
int count;              number of characters scanned
char *p;                input string
int *r;                 result integer
```

DESCRIPTION

Converts a hexadecimal string into an integer. The process terminates only when a non-hex character is encountered. Valid hex characters are Ø-9, A-F, and a-f.

RETURNS

count = Ø if input string does not begin with a hex digit
= number of characters scanned

CAUTIONS

No check for overflow is made during the processing.

NAME

stcd_i -- convert decimal string to integer

SYNOPSIS

```
count = stcd_i(p, r);
int count;          number of characters scanned
char *p;            input string
int *r;             result integer
```

DESCRIPTION

Converts a decimal string into an integer. The process terminates when a non-decimal character is found. Valid decimal characters are Ø-9. The first character may be + or -.

RETURNS

count = Ø if input string does not begin with a decimal digit
= number of characters scanned

CAUTIONS

No check for overflow is made during the processing.

NAME

stpblk -- skip blanks (white space)

SYNOPSIS

```
q = stpblk(p);
char *q;            updated string pointer
char *p;            initial string pointer
```

DESCRIPTION

Advances the string pointer past white space characters (space, tab, or newline).

RETURNS

q = updated string pointer (advanced past white space)

CAUTIONS

Must be declared char *, as the stp prefix indicates.

NAME

stpsym -- get a symbol from a string

SYNOPSIS

```
p = stpsym(s, sym, symlen);
char *p;            points to next character in s
char *s;            input string
char *sym;          output string
int symlen;         sizeof(sym)
```

DESCRIPTION

Breaks out the next symbol from the input string. The first character of the symbol must be alphabetic (upper or lower case), and the remaining characters must be alphanumeric. Note that the pointer is not advanced past any initial white space in the input string. The output string is the null-terminated symbol.

RETURNS

p = pointer to next character (after symbol) in input string

CAUTIONS

Must be declared **char ***, as the **stp** prefix indicates. If no valid symbol characters are found, **p** will equal **s**, and **sym** will contain an initial null byte.

NAME

stptok -- get a token from a string

SYNOPSIS

```
p = stptok(s, tok, toklen, brk);
char *p;          points to next char in s
char *s;          input string
char *tok;        output string
int toklen;       sizeof(tok)
char *brk;        break string
```

DESCRIPTION

Breaks out the next token from the input string. The token consists of all characters in s up to but not including the first character that is in the break string. In other words, the break string defines a list of characters which cannot be included in a token. Note that the pointer is not advanced past any initial white space characters in the input string. The output string is the null-terminated token.

RETURNS

p = pointer to next character (after token) in input string

CAUTIONS

Must be declared char *, as the stp prefix indicates. If no valid token characters are found, p will equal s, and tok will contain an initial null byte.

NAME

stpchr/strchr/strrchr -- find specific character in string

SYNOPSIS

```
p = stpchr(s, c);
p = strchr(s, c);
p = strrchr(s, c);
char *p;        points to c in s (or is NULL)
char *s;        points to string being scanned
char c;         character to be located
```

DESCRIPTION

The stpchr and strchr functions scan the specified string to find the first occurrence of the specified character, while the strrchr function scans for the last occurrence of the character. In either case, a NULL pointer is returned if the character is not found in the string.

RETURNS

```
p = NULL if c not found in s
  = pointer to first c found in s (stpchr, strchr)
  = pointer to last c found in s (strrchr)
```

CAUTIONS

These functions must be declared char *.

NAME

stpbrk/strpbrk -- find break character in string

SYNOPSIS

```
p = stpbrk(s, b);
p = strpbrk(s, b);
char *p;            points to element of b in s
char *s;            points to string being scanned
char *b;            points to break character string
```

DESCRIPTION

These functions scan the specified string to find the first occurrence of a character from the break string b. In other words, b is a null-terminated list of characters being sought. If the terminator byte for s is hit first, a NULL pointer is returned.

RETURNS

```
p = NULL if no element of b is found in s
  = pointer to first element of b in s (from left)
```

CAUTIONS

These functions must be declared char *.

NAME

strspn/strcspn/stcis/stcisn -- find longest initial span

SYNOPSIS

```
length = strspn(s, b);
length = strcspn(s, b);
length = stcis(s, b);
length = stcisn(s, b);
int length;              span length in bytes
char *s;                 points to string being scanned
char *b;                 points to character set string
```

DESCRIPTION

These functions compute the number of characters at the beginning (left) of s that come from a specified character set. For strspn and stcis, the character set consists of all characters in b, while for strcspn and stcisn, the character set consists of all characters not in b.

RETURNS

length = number of characters from the specified set which appear at the beginning (left) of s

NAME

stcarg -- get an argument

SYNOPSIS

```
length = stcarg(s, b);
int length;          number of bytes in argument
char *s;             text string pointer
char *b;             break string pointer
```

DESCRIPTION

Scans the text string until one of the break characters is found or until the text string ends (as indicated by a null character). While scanning, the function skips over partial strings enclosed in single or double quotes, and the backslash is recognized as an escape character.

RETURNS

length = number of bytes (in s) in argument
= 0 if not found

NAME

stcpm -- pattern match (unanchored)

SYNOPSIS

```
length = stcpm(s, p, q);
int length;          length of matched string
char *s;             string being scanned
char *p;             pattern string
char **q;            points to matched string if found
```

DESCRIPTION

Scans the specified string to find the first substring that matches the specified pattern. The pattern is specified in a simple form of regular expression notation, where

?	matches any character
s*	matches zero or more occurrences of s
s+	matches one or more occurrences of s

The backslash is used as an escape character (to match one of the special characters ?, *, or +). The scan is not anchored; that is, if a matching string is not found at the first position of s, the next position is tried, and so on. A pointer to the first matching substring is returned at *q.

RETURNS

```
length = Ø if no match
       = length of matching substring, if successful
```

CAUTIONS

Note that the third argument must be a pointer to a character pointer, since this function really returns two values: a pointer to, and the length of, the first matching substring.

NAME

stcpma -- pattern match (anchored)

SYNOPSIS

```
length = stcpma(s, p);
int length;          length of matching string
char *s;             string being scanned
char *p;             pattern string
```

DESCRIPTION

Scans the specified string to determine if it begins with a substring that matches the specified pattern. See the description of stcpm for a specification of the pattern format.

RETURNS

```
length = Ø if no match
       = length of matching substring if successful
```

NAME

stspfp -- parse file pattern

SYNOPSIS

```
error = stspfp(p, n);
int error;              return code:  -1 if error
char *p;                file name string
int n[16];              node index array
```

DESCRIPTION

Parses a file name pattern which consists of node names separated by slashes. Each slash is replaced by a null byte, and the beginning index of that node is placed in the index array. For example, the pattern /abc/de/f has three nodes, and their indexes are 1 for abc, 5 for de, and 8 for f. Note that the leading slash, if present, is skipped. Note also that a slash that is part of a node name (usually unwise) must be preceded by a backslash. The last entry in the node array n is set to -1 (in the example above, this causes n[3] to be -1).

RETURNS

error = Ø if successful
= -1 if too many nodes or other error

3.3.4 Utility Macros

The standard I/O header file stdio.h defines three general utility macros which are useful in working with arithmetic objects. They are:

max(a,b)	returns the maximum of a and b
min(a,b)	returns the minimum of a and b
abs(a)	returns the absolute value of a

Several important restrictions must be noted.

First, since these are macros which use the conditional operator, arguments with side effects (such as function calls or increment or decrement operators) cannot be used, and the address-of operator cannot be applied to these "functions". Second, beware of using the macro names in declarations such as

```
int min;
```

because the compiler will try to expand min as a macro, and an error message complaining of invalid macro usage will be generated. Third, only arithmetic data items should be used as arguments to these macros; max and min should be supplied two arguments of the same data type, although conversion will be performed if necessary.

3.4 Mathematical Functions

The functions described here include a large proportion of the floating point math functions that are usually provided with UNIX. Detailed specifications are given in the following manual pages. Note that the header files math.h and limits.h should generally be included when using these functions.

NAME

exp/log/log10/pow/sqrt -- exponential/logarithmic functions

SYNOPSIS

```
r = exp(x);          compute exponential function of x
r = log(x);          compute natural log of x
r = log10(x);        compute base 10 log of x
r = pow(x,y);        compute x to power y
r = sqrt(x);         compute square root of x

double r;            result
double x,y;          arguments
```

DESCRIPTION

These functions return the result of the indicated exponential, logarithmic, and power computations on double operands.

For log, log10, and sqrt, the x argument must be positive, and for pow, the y argument must be an integer if x is negative.

CAUTIONS

These functions must be declared double, which can be accomplished simply by including math.h. Also note that constant arguments must be expressed as floating constants such as 3.0 instead of 3.

NAME

sin/cos/tan/asin/acos/atan/atan2 -- trigonometric functions

SYNOPSIS

```
x = sin(r);          compute sine of r (r in radians)
x = cos(r);          compute cosine of r
x = tan(r);          compute tangent of r
r = asin(x);         compute arcsine of x
r = acos(x);         compute arccosine of x
r = atan(x);         compute arctangent of x
r = atan2(y,x);      compute arctangent of y/x

double r;            result
double x,y;          arguments
```

DESCRIPTION

The sin, cos, and tan functions compute the normal trigonometric functions of angles expressed in radians.

The asin function computes the inverse sine and returns a radian value in the range -PI/2 to +PI/2.

The acos function computes the inverse cosine and returns a radian value in the range Ø to PI.

The atan function computes the inverse tangent and returns a radian value in the range -PI/2 to +PI/2.

The atan2 function computes the inverse sine of y/x and returns a radian value in the range -PI to +PI.

CAUTIONS

These function must be declared double, which can be accomplished simply by including math.h.

NAME

sinh/cosh/tanh -- hyperbolic functions

SYNOPSIS

```
x = sinh(y);        compute hyperbolic sine
x = cosh(y);        compute hyperbolic cosine
x = tanh(y);        compute hyperbolic tangent

double x;           result
double y;           argument
```

DESCRIPTION

These functions compute and return the value of the indicated hyperbolic functions.

CAUTIONS

These functions must be declared double, which can be accomplished simply by including math.h.

NAME

rand,srand -- simple random number generation

SYNOPSIS

```
x = rand();
srand(seed);

int x;              random number
unsigned seed;      random number seed
```

DESCRIPTION

The rand function returns pseudo-random numbers in the range from Ø to the maximum positive integer value. At any time, srand can be called to reset the number generator to a new starting point. The initial default seed is 1. See the description of drand for more sophisticated random number generation.

NAME

drand -- generate random numbers

SYNOPSIS

```
x = drand48();      generate double (internal seed)
x = erand48(y);     generate double (external seed)

z = lrand48();      generate positive long (internal seed)
z = nrand48(y);     generate positive long (external seed)

z = mrand48();      generate long (internal seed)
z = jrand48(y);     generate long (external seed)

srand48(z);         set high 32 bits of internal seed
p = seed48(y);      set all 48 bits of internal seed
lcong48(k);         set linear congruence parameters

double x;           double precision random number
short y[3];         48-bit seed supplied by caller
long z;             long integer random number
short *p;           pointer to internal seed array
short k[7];         linear congruence parameters array
```

DESCRIPTION

These functions generate pseudo-random numbers using the linear congruential algorithm and 48-bit integer arithmetic. The normal versions (drand48, lrand48, mrand48) utilize an internal 48-bit storage area for the seed value. Special versions (erand48, nrand48, jrand48) are provided for cases where several seeds are in use at the same time, in which case the user provides the seed storage areas.

The drand48 and erand48 functions return values uniformly distributed over the interval from 0.0 up to but not including 1.0.

The lrand48 and nrand48 functions return non-negative long integers uniformly distributed over the interval from 0 to 2**31-1.

The mrand48 and jrand48 functions return signed long integers uniformly distributed over the interval from -2**31 to 2**31-1.

The srand48 and seed48 functions allow initialization of the internal 48-bit seed value to something other than the defaults. For srand48, the specified long value is copied into the high 32 bits of the seed, and the low 16 bits are set to 0x330e. For seed48, the entire 48-bits are loaded

from the specified array, and the function returns a pointer to the internal seed array.

The lcong48 function allows a much more intricate initialization of the linear congruential algorithm. The algorithm is of the form:

```
X[n+1] = (a * X[n] + c) mod m
```

where m is 2**48 and the default values for a and c are Øx5deece66d and Øxb, respectively. The array passed to lcong48 contains the value for X[n] in k[Ø] to k[2], the value for a in k[3] to k[5], and the value for c in k[6]. When seed48 is called, a and c are reset to their original default values.

CAUTIONS

The functions drand48 and erand48 must be declared double; the functions lrand48, nrand48, mrand48, and jrand48 must be declared long; and the seed48 function must be declared short *.

NAME

ceil/fabs/floor/fmod/frexp/ldexp/modf -- float conversions

SYNOPSIS

```
x = ceil(y);          get ceiling integer
x = fabs(y);          get absolute value
x = floor(y);         get floor integer
x = fmod(y,z);        get mod value
x = frexp(y,p);       split into mantissa and exponent
x = ldexp(y,i);       load exponent
x = modf(y,p);        split into integer and fraction

double x;             result
double y,z;           operands
int i;                binary exponent value
double *p;            for return of additional value
```

DESCRIPTION

These functions convert floating point numbers into various other forms.

The floor and ceil functions return the integer values that are just below and just above the specified value, respectively.

The fmod function returns y if z is zero. Otherwise, it returns a value that has the same sign as y, is less than z, and satisfies the relationship

```
y = i * z + x
```

where i is an integer.

The frexp function splits y into its mantissa and exponent parts. The exponent is placed into the area pointed to by p, while the mantissa is returned by the function.

The ldexp function returns y * (2 ** i).

The modf function returns the fractional part of y with the same sign as y and places the integer portion into the area pointed to by p.

CAUTIONS

These functions must be declared double, which can be accomplished simply by including math.h.

NAME

atof/atoi/atol -- simple ASCII conversions

SYNOPSIS

```
x = atof(p);        ASCII to floating point
i = atoi(p);        ASCII to integer
l = atol(p);        ASCII to long integer

double x;           double precision result
int i;              integer result
long l;             long integer result
char *p;            pointer to ASCII string
```

DESCRIPTION

These functions skip over any leading white space (i.e., blanks, tabs, and newlines) and then perform the appropriate conversion. The conversion stops at the first unrecognized character, and no check is made for overflow.

For atof, the ASCII string may contain a decimal point and may be followed by an e or an E and a signed integer exponent. For all functions, a leading minus sign indicates a negative number. White space is not allowed between the minus sign and the number or between the number and the exponent.

CAUTIONS

The function atof must be declared double, and the function atol must be declared long.

NAME

strtol -- convert ASCII to long integer

SYNOPSIS

```
r = strtol(s,p,base);

long r;         result
char *s;        string to be scanned
char **p;       returns pointer to terminating character
int base;       conversion base
```

DESCRIPTION

Converts an ASCII string into a long integer, using the specified number base for the conversion. Leading white space (blanks, tabs, and newlines) is skipped, and the conversion proceeds until an unrecognized character is hit. The pointer to the unrecognized character is returned in p. If no conversion can be performed, p will contain s, and the result will be Ø.

The conversion base can be in the range from Ø to 36. If it is non-zero, then the ASCII string may contain digit characters from Ø through 9 and from the letter A through as many letters as necessary, with no distinction made between upper and lower case. For example, if base is 13, then the allowable digit characters are Ø through 9 and A,B, and C or a, b, and c. If base is 16, then a leading "Øx" or "ØX" may appear in the string.

If base is Ø, then the leading characters of the string are examined to determine the conversion base. A leading "Ø" indicates octal conversion (base 8), while a leading "Øx" or "ØX" indicates hexadecimal conversion (base 16). A leading digit from 1 to 9 indicates decimal conversion (base 1Ø).

CAUTIONS

Must be declared as returning long.

NAME

ecvt -- convert floating point to ASCII

SYNOPSIS

```
p = ecvt(value,ndig,dec,sign);

char *p;            pointer to ASCII string
double value;       value to convert
int ndig;           number of digits in string
int *dec;           returns position of decimal point
int *sign;          non-zero if negative
```

DESCRIPTION

Converts the specified value into a null-terminated ASCII string containing the specified number of digits. The integer pointed to by dec will then contain the relative location of the decimal point, with a negative value meaning that the decimal is to the left of the returned digits. The actual decimal point character is not included in the generated string.

CAUTIONS

The pointer returned points to a static array which is overwritten by each call to ecvt; thus, it should be copied elsewhere if necessary. The function must be declared as returning char *.

NAME

matherr -- handle math function error

SYNOPSIS

```
code = matherr(x);

int code;                   non-zero for new return value
struct exception *x;        math exception block
```

DESCRIPTION

This function is called whenever one of the other math functions detects an error. Upon entry, it receives the exception block that describes the error in detail. This structure is defined in math.h, as follows:

```
struct exception
     {
     int type;        error type
     char *name;      name of function having error
     double arg1;     first argument
     double arg2;     second argument
     double ret;      proposed return value
     };
```

The error type names defined in math.h are:

```
DOMAIN     =>   domain error
SING       =>   singularity
OVERFLOW   =>   overflow
UNDERFLOW  =>   underflow
TLOSS      =>   total loss of significance
PLOSS      =>   partial loss of significance
```

The standard version of matherr supplied in the library places the appropriate error number into the external integer errno, and returns zero. When matherr is called, the function that detected the error will have placed its proposed return value into the exception structure. The zero return code indicates that return value should be used.

Programmers may supply their own version of matherr, if desired. On particular errors, it may be desirable to cause the function detecting the error to return a value other than its usual default. This can be accomplished by storing a new return value in ret of the exception structure, and then returning a non-zero value from matherr, which forces the function to pick up the new value from the exception structure.

SECTION 4:
Program Generation for AmigaDOS

The Lattice 68000 C compiler can be used on Amiga's AmigaDOS operating system to generate programs to be executed on the Amiga 68000 processor. It accepts text files containing programs written in the C programming language and produces relocatable machine code in the Amiga format, suitable for use by the Amiga linker ALINK.

4.1 Module Compilation

The compiler is implemented as two executable files, LC1 and LC2. Each program performs a portion of the compilation process and must be invoked by separate commands; LC1 does not automatically load LC2 when it completes its processing. Normally, LC2 should be executed immediately after LC1 if there are no errors in the source file. The compilation process can be diagrammed as follows:

```
file.C -> LC1 -> file.Q
file.Q -> LC2 -> file.O
```

LC1 reads a C source file, whose name must end in the two characters .C, and (provided there are no fatal errors) produces an intermediate file of an identical name, except that it ends in the two characters .Q. LC2 reads an intermediate file created by LC1 and produces a binary file of the same name but ending in .O. The intermediate file is deleted by LC2 when it completes its processing. Each phase normally creates its output file in the same directory as the input file. Note that if a source file defines more than one function, so does its resulting object file. Individual functions cannot be broken out from the object file when a program is linked; see Section 4.3.2 for more information.

The object file produced by the compiler must be incorporated with other object files and with run-time support subroutines in order to produce an executable file. This can be accomplished by using the Amiga utility ALINK; instructions for linking are presented in Section 4.2.

4.1.1 Phase 1

The first phase of the compiler reads a C source file and produces an intermediate file of logical records called quadruples, or quads. See Section 4.3.1 for a more detailed discussion of the processing performed. The format of the command to invoke the first phase of the compiler is:

LC1 [>listfile] [options] filename <RETURN>

The various command line specifiers are shown in the order they must appear in the command. Required specifiers are shown in emphasized type; optional specifiers are shown enclosed in brackets.

>listfile Causes the first phase messages to be written to a specified file. These messages include the compiler sign-on message and any error or warning messages which may be generated. The full filename must be specified, including extension, if any. If the file already exists, it is truncated and reused. This option is useful for reviewing long lists of error messages.

options Compile time options are specified as a hyphen followed by a single letter; in some cases, additional text may be appended. The option letter may be supplied in either upper or lower case. Each option must be specified separately, with a separate hyphen and letter (that is, they cannot be combined as they can for certain UNIX programs). Current options include:

-b Informs the first phase that all static and external data is to be addressed using a base register, either A5 or A6, thus limiting the total size of static data objects to 64K bytes. Which address register will be used depends on whether the -f option is specified on LC2 (see Section 4.1.2 below); A5 is the default. This option must be used if position-independent code is desired.

-c[flags] Controls the various compatibility modes of the compiler, which allow it to accept source files compiled with a previous version of the Lattice compiler. Each flag is specified as a single letter in either upper or lower case; more than one flag may be attached to the -c, but no blanks are permitted (for example, -cusw). The flag letters recognized are:

c Allows comments to be nested; the default now is that comments do <u>not</u> nest, in accordance with the generally accepted convention.

d Allows the dollar sign ($) to be used as an embedded character in identifiers.

m Permits the use of multiple character constants (for example, 'XX').

s Causes the compiler to generate only one copy

of identical string constants. By default the compiler now generates unique copies of all string constants, even if they are identical.

u Forces all char declarations to be interpreted as unsigned char.

w Shuts off the warning generated for return statements which do not specify a return value inside an int function. (Functions which do not declare a value should be declared void.)

-d Causes debugging information to be included in the quad file. Specifically, line separator quads are interspersed with the normal quads. This allows the second phase to collect information relating input line numbers to program section offsets. If this option is used, the object file produced will contain line number/offset records, and can be processed by the object module disassembler to produce an intermixed source code and machine code listing (see Section 4.1.4 below). Note that the -d option does not affect the size of the function itself, only the object file.

-dsymbol
-dsymbol=value Causes the identifier "symbol" to be defined, as if the compiler had encountered a #define command for it. One of two forms of the option may be used. The first form merely defines the symbol with a null substitution text; the equivalent C statement is

```
#define symbol
```

The second form uses an equal sign to attach a substitution text "value"; its equivalent is

```
#define symbol value
```

Several definitions can be made in the same LC1 command; however, macros with arguments cannot be defined from the command line. This feature allows source files containing conditional compilation directives (#ifdef, #ifndef, #if, #else, #endif) to be used to produce different results without modifying the source file, simply by defining the appropriate symbol on the LC1 command.

-iprefix Specifies that #include files are to be searched for by prepending the filename with the string prefix, unless the filename in the #include

statement is already prefixed by a slash, backslash or period (/,\ or .) directory specifier. Up to 4 different -i strings may be specified in the same LC1 command. When an unprefixed #include filename is encountered, the current directory is searched first; then file names are constructed and searched for, using prefixes specified in -i options, in the same left-to-right order as they were supplied on the command line. No intervening blanks are permitted in the string following the i. Note that if a directory name is to be specified as a prefix, a trailing slash must be supplied (see examples below).

-l Forces alignment of all data elements except char and short to a byte offset evenly divisible by four. This option is provided in the event that code is to be generated for a later version of the 68000 series which may perform long operations more efficiently if the values are stored at modulo 4 addresses.

-n Causes the compiler to retain a maximum of 8 characters for all identifier symbols, including #define symbols. The default symbol retention length is 31 characters.

-oname Modifies the name of the output file (the .Q or quad file). If name ends with a backslash (\), colon (:) or slash (/), the output file name is formed by prepending the input filename (the .C file which is being compiled) with name. Any drive or directory prefixes originally attached to the input filename are discarded before the new prefix is added. If name does not end with one of these characters, it is used as the complete name of the output file. Note that no intervening blanks are permitted in the string following the o.

-p Causes the compiler to create a text file containing the logical output of the compiler's preprocessor. If this option is used, no quad file is created. The name of the text file created is formed by replacing the .C portion of the source file name with .P. This option may be used to examine the result of complex macros or conditional compilation directives.

-u Cancels all automatic symbol definitions for the current compilation. Certain #define symbols are normally pre-defined by the compiler (see below);

this flag cancels all of those definitions.

-x Changes the default storage class for external declarations (made outside the body of a function) from external definition to external reference. The usual meaning of an external declaration for which an explicit storage class is not present is to define storage for the object and make it visible in other files. The -x option causes such declarations to be treated as if they were preceded by the **extern** keyword, that is, the object being declared is present in some other file.

filename Specifies the name of the C source file which is to be compiled; this is the only command line field which must be present. The filename should be specified without the .C extension; the first phase supplies the extension automatically. Alphabetic characters may be supplied in either upper or lower case. Note that only files with a .C extension can be compiled; if some other extension is specified, the compiler ignores it and tries to find name.C. (**#include** files, on the other hand, must be fully specified with extensions.) The current directory is assumed unless another directory is specified, and the quad file is created in the same directory as the source file unless the -o option is used (see above).

Include files may be specified enclosed in double quotes ("filename") or angle brackets (<filename>); the two forms have exactly the same effect. The name between the delimiters is taken at face value; the extension must be specified if one is defined for the file. The usual convention is to use .H for all header files. Alphabetic characters in a file name may be specified in either upper or lower case. Note that the current directory is always searched first for #include files, and that prefixes specified in -i options are used only if the name on the #include line is not prefixed with a slash, backslash or period (/,\ or .).

As an assistance to conditional compilation, the compiler automatically #defines several symbols, which can be tested in #if, #ifdef, or #ifndef directives to select appropriate code sequences according to the target processor or operating system. Three symbols are always defined in the compiler:

```
#define M68000 1
#define AMIGA 1
#define SPTR 1
```

If the -d flag was specified (as "-d", not "-dsymbol"), the

following symbol is defined:

```
#define DEBUG 1
```

The automatic definition of these symbols can be prevented by use of the -u option, which cancels all of the above definitions.

EXAMPLES

LC1 xyzfile -idf0:include/ xyzfile

This command executes the first phase of the compiler using file XYZFILE.C as input, creating file XYZFILE.Q in the current directory. Any #include files not found in the current drive/directory will be searched for in the directory DF0:INCLUDE. Note the trailing slash on the prefix attached to the -i flag; it is not automatically assumed by the compiler.

LC1 XYZ -odf1: -x XYZ

This command executes the first phase of the compiler using file XYZ.C as input, creating file XYZ.Q on DF1; it causes all external declarations without a storage class to be interpreted as extern declarations.

LC1 >tns.err -ccuw tns

This command executes the first phase of the compiler using file TNS.C as input, creating file TNS.Q on the currently logged-in disk; it creates a file TNS.ERR to contain all of the messages generated by the compiler. The compiler will allow nested comments, interpret all char declarations as unsigned char, and suppress warning messages for return statements with no return value in int functions.

4.1.2 Phase 2

The second phase of the compiler reads a quad file created by the first phase and creates an object file in the Amiga format. See Section 4.3.2 for a more detailed discussion of the processing performed. The format of the command to invoke the second phase of the compiler is:

LC2 [options] **filename <RETURN>**

The various command line specifiers are shown in the order they must appear in the command. Required specifiers are shown in emphasized type; optional specifiers are shown in enclosed in brackets.

options Compile time options are specified as a hyphen followed by a single letter; in some cases, additional text may be appended. The option letter may be supplied in either upper or lower case. Each option must be specified separately, with a separate hyphen and letter (that is, they cannot be combined as they can for certain UNIX programs). Current options include:

-fn Specifies an address register to be used for the stack frame pointer. Only two values for "n" are allowed: 5 indicates that register A5 is to be used, 6 that register A6 is to be used. The address register used if the -f option is not specified is A6. If the **-b** option is specified on LC1, whichever of these two registers is not used as the stack frame pointer is used as the base register for addressing static and external data.

-oname Modifies the name of the output file (the .O file). If name ends with a backslash (\), colon (:) or slash (/), the output file name is formed by prepending the input filename (the .Q file which is being compiled) with name. Any drive or directory prefixes originally attached to the input filename are discarded before the new prefix is added. If name does not end with one of these characters, it is used as the complete name of the output file. Note that no intervening blanks are permitted in the string following the o.

-p Causes the code generator to insert a special instruction known as a "stack probe" at the entry of each function for which code is generated. This option is used when generating programs to run under operating systems which provide run-time memory management of stack allocation, such as Microsoft's Xenix operating system.

-r Forces all function calls to use PC-relative addressing, thus limiting the range of function calls to plus or minus 32K. This option must be used if position-independent code is desired.

-s Adds section names to the object file. The Amiga linker will merge all sections (hunks) with identical names. Normally, the object file contains unnamed sections (hunks) which are not merged with other sections; such unnamed sections may be placed in memory at widely different addresses when the final program is loaded. The sections in

all modules compiled with the -s flag, on the other hand, will be loaded in contiguous memory. This may result in a faster load time for the program, but it may also prevent it from loading under certain conditions. The -s flag is useful for debugging purposes to force certain modules to be loaded together.

filename Specifies the name of the intermediate file from which code is to be generated; this is the only command line field which must be present. This intermediate file is a quad file with a .Q extension, created by the first phase of the compiler. The file name should be specified without the .Q extension; the second phase supplies the extension automatically. Alphabetic characters may be supplied in either upper or lower case. The current directory is assumed unless another directory is specified, and the object file is created in the same directory as the quad file unless the -o option is used (see above).

EXAMPLES

LC2 -odf0: u790

This command executes the second phase of the compiler using file U790.Q as input, causing the file U790.O to be created on DF0.

LC2 -f5 test4

This command executes the second phase of the compiler using file TEST4.Q as input, causing the file TEST4.O to be created. Address register A5 will be used as the stack frame pointer in the code generated.

4.1.3 LC Command (Compiler Driver)

Supplied with the compiler is a program called LC that uses the program load capabilities of AmigaDOS to call the two compiler passes repeatedly for multiple compilations. The format of the command to invoke this program is:

LC [options] files

where **options** is a list of compiler options and **files** is a list of filenames, separated by white space. Just as with LC1 and LC2, the filenames should be specified without the .C extensions.

In general, the options are the same as for the LC1 and LC2 commands, except where LC1 and LC2 use the same option letter to mean different things. This occurs only for the -o option, and

is resolved as follows:

-qx Specifies prefix for quad files; same as LC1 -o.

In other words, the quad file prefix is indicated using -q instead of -o.

By default, LC directs the quad file to RAM:, allowing compilations to proceed more quickly than if the quad file is created on disk. However, on systems with less than 512K of memory, there may not be enough memory to contain the quad file. In this instance, the -q option should be used to cause the quad file to be generated on disk.

LC allows a blank between an option letter and the string that follows it, as in

LC -d xyz file

This is compatible with UNIX, but causes a problem if the -d item is just before the file name part of the command and was intended to indicate debugging mode instead of defining a symbol, as in:

LC -d program

The symbol "program" will be #defined instead of being treated as a file name to be compiled. This problem can be avoided through use of the UNIX convention of ending the options with a single dash:

LC -d - program

Use of the -V option will cause LC to display the command lines used to execute each phase of the compiler.

EXAMPLES

LC -i/include/ -p ptrsub klax portq

This command compiles the files PTRSUB.C, KLAX.C, and PORTQ.C, producing PTRSUB.O, KLAX.O, and PORTQ.O, if successful compilations are made for each file. Any #include files not found in the current directory will be searched for in /INCLUDE, and the code for any functions generated will include a stack probe instruction on entry.

4.1.4 Object Module Disassembler

The object module disassembler (OMD) provides a listing of the machine language instructions generated for a particular C source module. If the module is compiled with the -d option so that

line number/offset information is included in the object file, the disassembler utility can produce a listing with interspersed source code lines. This listing can then be used in association with the link map for the program to determine the exact location of the code generated to perform specific statements.

The format of the command to invoke the object module disassembler is

OMD [>listfile] [options] objfile [textfile]

The various command line specifiers are shown in the order they must appear in the command. Optional specifiers are shown enclosed in brackets.

>listfile The first option is used to direct the listing produced by OMD to a specified file or device. If this option is omitted, the listing output is written to the user's console.

options Two override options can be specified; each consists of a hyphen followed by a single letter which indicates the value to be overridden, and a string of decimal digits specifying the override value. The option letter may be specified in either upper or lower case. There must be no embedded blanks in any single option, but each must be specified as a separate field. The valid options are:

-Xnnn Overrides the default maximum number of external items which can be processed by OMD; this number applies separately to both external definitions and external references. nnn specifies a decimal number of external items which can be processed. The default value is 200.

-Lnnn Overrides the default size for the line number and offset information tables. These tables are used only if the object file was produced with the -d option; line number/offset information from the file is placed in these tables. The default size (which defines the maximum number of line number/offset pairs which can be processed) is 100.

objfile Specifies the name of the object file, produced by the compiler, which is to be processed by OMD. The full name including the .O extension must be specified.

textfile Specifies the name of a C source code file which is to be listed along with the disassembled instructions. If this option is present, the object file must have been compiled using the -d option for the LC1 command. The

full name including the .C extension must be specified.

OMD processes only a single object module. The entire module is read and loaded into memory before the listing is generated. The memory needed to contain the program and data sections of the module is automatically allocated, since the sizes of the files are contained in the file.

If the textfile option is used, only the source text from the specified file is listed; if it refers to any #include files, they will not be listed. Some limitations of the textfile option should be noted. First, the code generated for the third portion of for statements is placed at the bottom of the loop; that code will appear in front of the next statement after the end of the loop. Second, the compiler tends to defer storing registers until the last possible moment, so that the code shown for assignment statements often consists merely of loading values into registers; the registers will be stored later. Finally, the code generated for entry to a function will often be displayed in front of the source lines defining that function. Thus, inspection of the surrounding code may be necessary to determine the actual code generated for a source file construct.

EXAMPLES

OMD QRS.O

This command disassembles the object module QRS.O and writes the listing to the user's console.

OMD >TEMP.LST -X400 XYZ.O XYZ.C

This command disassembles the object module XYZ.O and writes the listing to the file TEMP.LST; it causes the source code lines from XYZ.C to be placed in the listing, provided that line number and offset information is present in the object file. It also provides for a maximum number of 400 external items (same limit for both external definitions and external references).

ERROR MESSAGES

A variety of error conditions are detected by the object module disassembler; all cause early termination of the output file and result in the writing of an appropriate error message to stderr. These messages are self-explanatory for the most part. If one of the run-time-specifiable options is not sufficiently large, the error message will indicate the specific option which was not large enough; for example, if the module defines too many external definitions or references, the message

External items overflow

will be produced.

4.2 Program Linking

After all of the component source modules for a program have been compiled, they must be linked together to produce an executable load module, or program file. This step is necessary for several reasons. First, the object file produced by the second phase of the compiler is not is a state suitable for execution. Second, most programs make use of external functions not defined in the same module; before such programs can execute, they must be "connected" with those other functions. The functions may be defined by the user, in which case they must be compiled and available as object files during linking, or they may be defined in the library supplied with the compiler. Note that the compiler generates internal function calls to perform certain operations (see Section 5.1).

Although the usual concept of linking involves external function calls, C also permits functions to access data locations defined in other modules. This kind of reference is possible because the external linkage mechanism supported by the object code associates an external symbol with a memory location; this symbol is the identifier used to declare the object in a C program. The programmer must be careful to declare an object with the same attributes in both the module which defines it and the module which refers to it, because linking does not verify the type of reference made -- it simply connects memory references using external symbols. The use of include files for common external declarations will usually prevent this kind of error.

The linking process in a general sense requires that all the components of a program be specified, either directly or indirectly, as input to the linker. Three types of input are required.

1. The start-up file C.O must be specified as the first module included by the linker. This file defines the AmigaDOS entry point for all C programs using the Lattice C compiler.

2. Functions generated by the user must be specified as additional modules to be included. These modules include the main module, as well as any additional functions defined in other source modules.

3. The two files LC.LIB and AMIGA.LIB must be specified as libraries to be searched during linking.

In the case of the linker supplied with AmigaDOS, these inputs

are specified by:

1. Making C.O the first module after the FROM keyword **ALINK** command.

2. Including the names (with .O extensions) of the user's object files on the **ALINK** command, along with the C.O start-up module after the FROM keyword.

3. Including **LC.LIB** and **AMIGA.LIB** after the LIBRARY keyword on the **ALINK** command.

Note that for step (2), one of the files included in the FROM list must be the main module, i.e., it must define the function main.

If the linker cannot find one of the .O files mentioned on the **ALINK** command, it will stop processing without creating an executable file. Another error condition can arise if the linker cannot find all the external items referred to in the .O files specified. In this case the message **Linker: unresolved external references:** will be generated by the linker, followed by a list of the external names which were not defined and the modules which referenced them. No attempt to execute a program with unresolved externals should be made unless it is certain that the missing functions will never be called.

EXAMPLE

```
ALINK FROM C.O,XYZ.O,QRS.O TO CPROG LIBRARY LC.LIB,AMIGA.LIB
```

This command executes the linker, producing CPROG as an executable program, and causes the files XYZ.O and QRS.O to be included in the program. The start-up module C.O is named first in the FROM list, and the files LC.LIB and AMIGA.LIB are specified as libraries to be searched.

4.3 Compiler Processing

The Lattice C compiler under AmigaDOS is implemented as two separately executable programs, each performing part of the compilation task. This section discusses the structure of the compiler in general terms, and describes the processing performed by both phases. Special sections are devoted to a discussion of the topics of error processing and code generation.

4.3.1 Phase 1

The first phase of the compiler performs all pre-processor functions concurrently with lexical and syntactical analysis of the input file. It generates the symbol tables, which contain

information about the various identifiers in the program, and produces an intermediate file of logical records called quadruples, which represent the elementary actions specified by the program. When the entire source program has been processed (assuming there are no fatal errors), the intermediate file (also called the quad file) is reviewed, and locally common sub-expressions are detected and replaced by equivalent results. Then selected symbol table information is written to the quad file, for use by the second phase. The first phase is thus very active as far as disk I/O is concerned. Generally, if the disk activity stops for more than a few seconds, it is reasonable to assume that the compiler has failed. See Appendix B for the compiler error reporting procedure if this happens.

When the first phase begins execution, it writes a sign-on message to the standard output which identifies the version of the compiler which is being executed. On AmigaDOS the first phase returns an exit code of zero if no errors were detected, and a code of one otherwise. See Section 4.3.3 for more information about error processing. Note that the quad file is deleted if any fatal errors are detected, but not until the source file has been completely processed. Aborting the first phase before it has terminated can therefore result in the existence of an invalid quad file which should have been deleted.

If the -p option is specified, the first phase of the compiler does not create a quad file; instead, it creates a text file with the logical output of the preprocessor. In this case, no syntactic checking of the file is performed, and only error messages relating to the preprocessor are generated.

4.3.2 Phase 2

The second phase of the compiler scans the quad file produced by the first phase, and produces an object file in the Lattice format. This object code supports all of the necessary relocation and external linkage conventions needed for C programs (see Section 5.3 for details). A logical section of code text specifying the machine language instructions which make up the executable portion of the program is generated first, followed by a section of data-defining text for all static items. Unlike the first phase, the code generator is not always actively performing disk I/O. Each function is constructed in memory before its object code is generated, so that there may be fairly sizable pauses during which no apparent disk activity is taking place. In general, these delays should not last more than several seconds. If no activity occurs for more than about 30 seconds, the compiler has probably failed; see Appendix B for information about reporting compiler problems.

When the second phase begins execution, it writes a sign-on message to the standard output which identifies the version of

the code generator which is being executed. When code generation is complete, the second phase writes a message of the form

Module size P=pppp D=dddd U=uuuu

to the standard output (usually the user's console). pppp indicates the size in bytes of the program or executable portion of the module generated, dddd the size in bytes of the initialized data section, and uuuu the size in bytes of the uninitialized data section; all values are given in hexadecimal. These sizes include the requirements for all of the functions included in the original source file. Note that the sizes define the amount of memory required for the module once it is loaded (as part of a program) into memory; the .O file requires more space (or less) because it contains additional relocation information, and does not represent uninitialized static data directly.

As noted in the introduction to Section 4.1, the code generator produces a single .O module for a given source module, regardless of how many functions were defined in that module. These functions (if more than one is defined) cannot be separated at link time; if any one of the functions is needed, all of them will be included. Functions must be separated into individual source files and compiled to produce separate object modules if it is necessary to avoid this collective inclusion.

4.3.3 Error Processing

All error conditions (with the exception of internal compiler errors) are detected by the first phase. As soon as the first fatal error is encountered, the compiler stops generating quads; it then deletes the quad file just before it terminates execution. This prevents the second phase from attempting to generate code from an erroneous quad file. Note that under AmigaDOS the compiler returns a zero if no errors are detected, and a one otherwise. When the compiler detects an error in an input file, it generates an error message of the form:

filename line **Error** nn: descriptive text

where filename is the name of the current input file (which may not be the original source file if #include files are used); line is the line number, in decimal, of the current line in that file; nn is an error number identifying the error; and descriptive text is a brief description of the error condition. (Appendix A provides expanded explanations for all error and warning messages produced by the compiler.) All error messages are written to the standard output, which is normally the user's console but can be directed to a file if desired (see Section 4.1.1). A message similar to the one above but with the text Warning instead of Error is generated for non-fatal errors; in this case, generation of the quad file continues normally. In some cases, an error

message will be followed by the additional message:

Execution terminated

which indicates that the compiler was too confused by the error to be able to continue processing. The compiler uses a very simple error recovery scheme which may sometimes cause a single error to induce a succession of subsequent errors in a "cascade" effect. In general, the programmer should attempt to correct the obvious errors first and not be overly concerned about error messages for apparently valid source lines (although all lines for which errors are reported should be checked).

Error messages which begin with the text CXERR are internal compiler errors which indicate a problem in the compiler itself. See Appendix B for the compiler error reporting procedure. The compiler generates a few other error messages that are not numbered; they are usually self-explanatory. The most common of these is the Not enough memory message, which means that the compiler ran out of working memory.

SECTION 5:
68000 Code Generation

Any processor with a sufficiently rich instruction set allows implementation of high-level language constructs in a variety of ways, and the 68000 is no exception. This section presents the general strategy used by the Lattice compiler in generating code for the 68000, with a view toward clarifying the machine-dependent aspects of the language, the compiler's choice of machine language instructions, and the interface to user-written assembly language modules.

5.1 Machine Dependencies

The C language definition does not completely specify all aspects of the language; a number of important features are described as machine-dependent. This flexibility in some of the finer details permits the language to be implemented on a variety of machine architectures without forcing code generation sequences that are elegant on one machine and awkward on another. This section describes the machine-dependent features of the language as implemented on the 68000 series. See Section 2 of the manual for a description of the machine-independent features of the Lattice implementation of the language.

5.1.1 Data Elements

The standard C data types are implemented according to the following descriptions. The only data elements which do not require alignment to a word offset are characters and character arrays; as noted in Section 4.1.1, this word alignment can be forced to a long word (four-byte) alignment for all objects larger than two bytes by a compile time option. In all cases, regardless of the length of the data element, the high order (most significant) byte is stored first, followed by successively lower order bytes. This scheme is consistent with the general byte ordering used on the 68000. The following table summarizes the characteristics of the data types:

Type	Length in Bits	Range
char	8	-128 to 127 (ASCII characters)
unsigned char	8	0 to 255
short	16	-32768 to 32767
unsigned short	16	0 to 65535
int	32	-2147483648 to 2147483647
unsigned int	32	0 to 4294967295
long	32	-2147483648 to 2147483647
float	32	+/- 10E-37 to +/- 10E38
double	64	+/- 10E-307 to +/- 10E308

char	defines an 8-bit unsigned integer. Text characters are generated with bit 7 reset, according to the standard ASCII format.
unsigned char	defines an 8-bit unsigned integer.
int long long int	all define a 32-bit signed integer.
unsigned or unsigned int unsigned long	all define a 32-bit unsigned integer.
short or short int	both define a 16-bit signed integer.
unsigned short	defines a 16-bit unsigned integer.
float	defines a 32-bit signed floating point number, with an 8-bit biased binary exponent, and a 24-bit fractional part which is stored in normalized form without the high-order bit being explicitly represented. The exponent bias is 127. This representation is equivalent to approximately 6 or 7 decimal digits of precision.
double or long float	defines a 64-bit signed floating point number, with an 11-bit biased binary exponent, and a 53-bit fractional part which is stored in normalized form without the high-order bit being explicitly represented. The exponent bias is 1023. This representation is equivalent to approximately 15 or 16 decimal digits of precision.

Pointers to the various data types and to functions are four bytes in length, and contain the absolute address of the first byte of the target object.

The total size of all objects declared within the same storage class is limited according to the particular class, as follows:

Storage Class	Maximum total size of objects declared
extern	1048575
static	1048575
auto	524287
formal	255

Note that aggregates (structures and unions) declared as formal

parameters do not contribute their actual size to the total storage for formals, but rather the size of a pointer. This is because aggregates passed by value are passed as a pointer which is used to move the contents of the aggregate to a local copy within the function; see Section 5.3.3.

5.1.2 External Names

External identifiers may be up to 31 characters in length in the default case; if the -n option is used on LC1, they will be truncated to 8. Upper and lower case letters are distinct, i.e., case is significant. A user may define external objects with any name that does not conflict with the following classes of identifiers:

_******* Certain library functions and data elements (defined in modules written in C) are defined with an initial underscore.

CX**** Run-time support functions (written in assembly language) which implement C language features such as long integer multiply and divide, floating point arithmetic, and the like are defined with CX as the first two characters.

The likelihood of collision with library definitions is remote, but users should be aware of these conventions and avoid applying these types of identifiers to external, user-defined functions and data.

5.1.3 Arithmetic Operations and Conversions

Arithmetic operations for the integral types (floating type operations are discussed in the next section) are generally performed by in-line code. Integer overflows are ignored in all cases, although signed comparisons correctly include overflow in determining the relative size of operands. Short integer division by zero generates a trap; long integer division by zero simply generates a result of zero. Division of negative integers causes truncation toward zero, just as it does for positive integers, and the remainder has the same sign as the dividend. Right shifts are arithmetic, that is, the sign bit is copied into vacated bit positions, unless the operand being shifted is unsigned; in that case, a logical (zero-fill) right shift is performed.

Function calls to library routines are generated only for long integer multiplication and division (both signed and unsigned).

Conversions are generated according to the "usual arithmetic conversions" described in Kernighan and Ritchie, and are generally trouble free. The following points should be noted:

1. char objects may be signed or unsigned in this implementation. Thus, sign extension may or may not be performed during expansion to int. Note that all char declarations may be forced to be interpreted as unsigned char by means of a compile time option; see Section 4.1.1.

2. Conversion of short to long causes sign extension, while conversion of unsigned short to long does not. The inverse operations simply truncate the result, which is undefined if its absolute value is too large to be represented.

3. Expansion of char and short operands to int may not be performed by the compiler if those operands only participate in operations with other operands of the same type, resulting in increased efficiency for sequences like

```
char a, b, c;
. . .
a = b + c;
```

   Note that expansion is, however, always performed for function call arguments.

4. Conversions from integral to floating types are fairly straightforward. The inverse conversions cause any fractional part to be dropped.

5. Conversion from float to double is well-defined, but the inverse operation may cause an underflow or overflow condition since double has a much larger exponent range. Considerable precision is also lost, though the fraction is rounded to its nearest float equivalent.

6. In general, the presence of any unsigned operand in an expression causes the result also to be unsigned.

5.1.4 Floating Point Operations

In accordance with the language definition, all floating point arithmetic operations are performed using double precision operands, and all function arguments of type float are converted to type double before the function is called. The formats used are identical to the 32-bit and 64-bit formats defined by the proposed IEEE standard for floating point representations. Legal floating point operations include simple assignment, conversion to other arithmetic types, unary minus (change sign), addition, subtraction, multiplication, division, and comparison for equality or relative size. Note that, in contrast to the signed

integer representations, negative floating point values are not represented in two's complement notation; positive and negative numbers differ only in the sign bit. This means that two kinds of zero are possible: positive and negative. All floating point operations treat either value as true zero and generally produce positive zero, whenever possible. Note that code which checks float or double objects for zero by type punning (that is, examining the objects as if they were int or some other integral type) may assume (falsely) negative zero to be not zero.

Floating point arithmetic and comparison operations are performed by generating calls to library routines written in assembly language. Floating point exceptions are processed by a library function called CXFERR that is called according to the following convention:

```
CXFERR(errno);
int errno;
```

where errno can be

```
1 = underflow
2 = overflow
3 = divide by zero
```

The standard version of CXFERR supplied in the libraries simply ignores all error conditions. A different version can be written (in either C or assembly language) to print out an error message and terminate processing, or take any other action. If CXFERR returns to the library function which called it, each exception is processed as follows:

Underflow	Sets the result equal to zero.
Overflow	Sets the result to plus or minus infinity.
Zerodivide	Sets the result equal to zero.

5.1.5 Bit Fields

Bit fields are fetched on a long word basis, that is, the entire word containing the desired bit field is loaded (or stored) even if the field is 16 bits or less in size. Bit fields are assigned from left to right within a machine word; the maximum field size is 31 bits. Bit fields are considered unsigned in this implementation; sign extension is not performed when the value of a field is expanded in an arithmetic expression. If a structure is declared

```
struct {
     unsigned x : 20;
     unsigned y : 9;
     unsigned z : 3;
     } a;
```

then a occupies a single 32-bit word, a.x resides in bits 31 through 12, a.y in bits 11 through 3, and a.z in bits 2, 1 and 0. Bit fields of only a single bit are tested and assigned constant values using the BTST, BSET, or BCLR instructions.

5.1.6 Register Variables

A register variable declaration may be accepted for any pointer or other data object with a size of no more than 4 bytes. Up to four pointers may be assigned to address registers starting with A5 down through A2; up to six simple data elements may be assigned to data registers starting with D7 down through D2. The registers are assigned in the same order in which they appear in the function declaration, with formal parameters being assigned first. Naturally, if A5 is used as a frame pointer via the -f option described in Section 4.1.2, it is not available for use as a register variable.

The use of register variables affects the entry sequence at the start of the function in which they are declared, by requiring an additional instruction to save the previous registers' values before they are used in the function. See Section 5.3.3 for more information.

5.2 General Code Generation Strategies

When the code for a function is buffered in memory before being written to the object file, branch instructions are not explicitly represented in the function image. Instead, they are represented by special structures denoting the type and target of each branch. When the function has been completely defined, the branch instructions are analyzed and several important optimizations are performed:

1. Any branch instruction that passes control directly to another branch instruction is re-routed to branch directly to the target location.

2. A conditional branch instruction that branches over a single unconditional branch is replaced by a single conditional branch instruction of the opposite sense.

3. Sections of code into which control does not flow are detected and discarded.

4. Each branch instruction is coded in the smallest possible machine language sequence required to reach the target location.

Most of these optimizations are applied iteratively until no further improvement is obtained.

The code generator also makes a special effort to generate efficient code for the switch statement. Three different code sequences may be produced, depending on the number and range of the case values.

1. If the number of cases is three or fewer, control is routed to the case entries by a series of test and branch instructions.

2. If the case values are all positive and the difference between the maximum and minimum case values is less than twice the number of cases, the compiler generates a branch table which is directly indexed by the switch value. The value is adjusted, if necessary, by the minimum case value and compared against the size of the table before indexing. This construction requires minimal execution time and a table no longer than that required for the type of sequence described in No. 3.

3. Otherwise, the compiler generates a table of [case value, branch address] pairs, which is linearly searched for the switch value.

All of the above sequences are generated in-line without function calls because the number of instruction bytes is small enough that little benefit would be gained by implementing them as library functions.

Aside from these special control flow analyses, the compiler does not perform any global data flow analysis or any loop optimizations. Thus, values in registers (except for register variables) are not preserved across regions into which control may be directed. The compiler does, however, retain information about register contents after conditional branches which cause control to leave a region of code. Throughout each section of code into which control cannot branch (although it may exit via conditional branches), values which are loaded into registers are retained as long as possible so as to avoid redundant load and store operations. The allocation of registers is guided by next-use information, obtained by analysis of the local block of code, which indicates which operands will be used again in subsequent operations. This information also assists the compiler, in analyzing binary operations, in its decision whether to load both operands into registers or to load one operand and use a memory reference to the other. Generally, the result of such an operation will be computed in a register, but sequences like

```
i += j;
```

will load the value of j into a register and compute the result directly into the memory location for i (but only if i is not

used later in the same local block of code).

The hardware registers D0 through D7 are used as general purpose accumulators, while A0 through A4 (and A5, if not used as a frame pointer) are used for pointer values, allowing access to indirect operands. Either A5 or A6 is used to address the current stack frame; see Section 5.3.3 for more information. The use of registers for register variables is described in Section 5.1.6.

5.3 Run-Time Program Environment

This section describes the run-time environment which is implicitly assumed by the 68000 code generator and its effect on the interface between C and assembly language. Some knowledge of the architecture of the 68000 processor and of basic object code and linkage concepts is required in order to understand much of the information presented.

The C programming language provides for three distinct classes of objects in memory: the instructions which make up the executable functions, the static data items which persist independently of any of the functions which refer to them, and the automatic data items which exist only while a function is invoked. Many implementations support, through library functions, an additional dynamic memory allocation facility which returns pointers to objects not explicitly declared. Because the 68000 processor has a linear address space, no special assumptions about the location of any of these components are needed. Thus, in the general case, functions and data may be placed anywhere in memory because of the following conventions:

(1) All function calls are generated using a JSR instruction with a 32-bit absolute address.

(2) All static and external data elements are accessed via 32-bit absolute addresses.

(3) All automatic data elements are allocated and accessed relative to address register A7; if the offset of a data element exceeds the directly addressable range of 32K bytes, it is accessed by transferring the frame pointer register to another address register and adding in the offset value.

Note that no stack overflow checking is performed by the generated code.

If the -r option on LC2 is used, function calls are generated using a JSR instruction with a 16-bit PC-relative offset. Thus, no function call can reach farther that 32767 bytes above or below itself. If the -b option of LC1 is used, static and external data elements are accessed using 16-bit offsets from an

address registar (A5 or A6). This limits the size of static and external data to a maximum of 65535 bytes.

These rather severe restrictions are necessary in order to produce true position-independent code. In addition, note that pointers cannot be initialized by static declarations such as

```
char *p = "string";
```

if position-independent code is desired; at compile time, the only way to initialize such an object is to generate a relocatable address reference that will result in an absolute address at run time. This problem illustrates another limitation of position-independent C code on the 68000: once the address of an object is computed and assigned to a pointer, the target object may not be moved to another location without destroying the validity of such a pointer.

5.3.1 Object Code Conventions

The object file created by the second phase of the compiler is in the standard Amiga format, and defines the instructions and data necessary to implement the module specified by the C source file; it also contains relocation and linkage information necessary to guarantee that the components will be addressed properly when the module is executed or referenced as part of a linked program. While the addressing conventions used by the code generator permit data and functions to be scattered randomly throughout memory, it is possible to force functions and data to be collected together at link time into two contiguous blocks. The object module produced by the compiler is designed to facilitate this grouping by placing functions and data into separate control sections (the Amiga documentation refers to them as hunks). At link time, all elements in the same named control section are placed in contiguous regions of memory. By default, the sections created by the compiler are not named; the -s flag is used to cause them to be named.

The code section (named "text" if the -s flag is used) is the control section which includes the instructions which perform the actions specified by any functions defined in the source file.

Two data sections are defined. The first (named "data" if the -s flag is used) is the control section which includes all explicitly initialized static data items which are defined in the source file, and the second (named "udata" if the -s flag is used) is the logical control section which specifies the size of all uninitialized static data items. Static data in this sense includes not only those data items explicitly declared static but also items declared outside the body of a function without an explicit storage class identifier. String constants are considered initialized static data, and are placed in the "data"

section. (Note that uninitialized static data items are not explicitly represented in the object file, although they are guaranteed to be zero at run time. Similarly, automatic data items are simply allocated on the stack at run time and are not explicitly defined in the object file.)

The net effect of these control section assignments (if the -s flag is used) is to force, at link time, all functions to be collected together and all static data items to be similarly combined. There are two advantages to this structure. In the event of a program error which addresses an array out of range, the effect is usually less catastrophic if it is only data (not instructions) which are destroyed. In addition, some processors may support memory management hardware which will allow protection or mapping of contiguous portions of memory; separating program and data portions of a program facilitates use of such capabilities.

5.3.2 Linkage Conventions

As noted in Section 5.1.2, external identifiers may be up to 31 characters in length, depending on whether the **-n** option was used when the module was compiled. The relocation information in the object file defines all external names as an unspecified type, that is, there is no set of attributes associated with the name; it is simply an address within the memory defined by the final load module. It is therefore an error to define two items with the same external name in the same program. It is the programmer's responsibility to prevent this occurrence, and also to make sure that modules refer to external names in a consistent way (i.e., a function should not refer to "xyz" as **short** when it is actually defined as **int** in some other module). External definition and reference from assembly language modules are discussed in Section 5.3.4.

5.3.3 Function Call Conventions

When a C function makes a call to another function, it first pushes the values of any arguments onto the stack and then makes a call to that function. For external functions, a JSR instruction with an absolute long address is used (resolved at link time); for functions defined in the same module, a BSR instruction is used (resolved at compile time). The arguments are pushed in right-to-left order because the stack grows downward on the 68000; this allows the called function to address the arguments in the natural left-to-right (low-address-to-high-address) order. Note that the C language definition requires all **char** and **short** arguments to be expanded to **int**, so that a minimum of four bytes is pushed for each argument. The first actions taken by the called function are as follows:

1. The value of the frame pointer register (either A5 or A6)

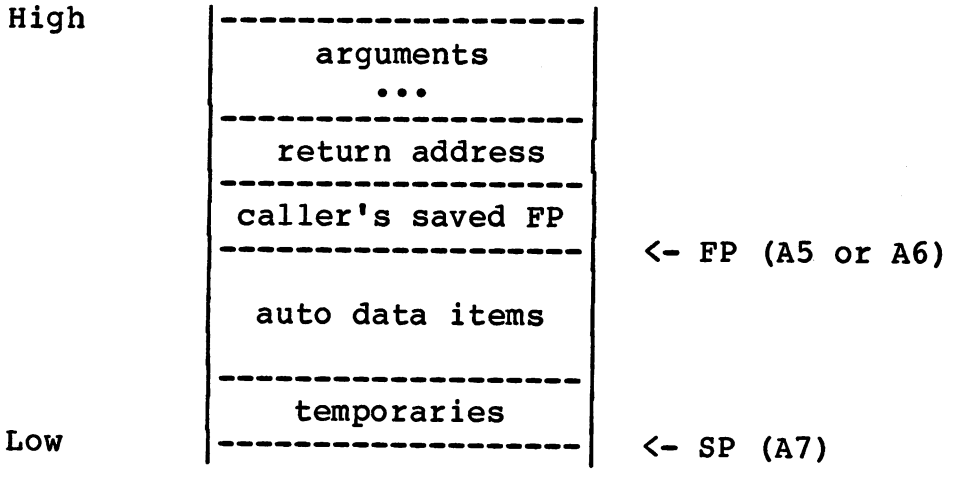
is saved on the stack, the stack pointer is transferred to the frame pointer register, and the stack pointer is then adjusted downward by the number of bytes of stack space required by the called function (this local storage is also called a stack frame). This sequence of operations is accomplished by a simple LINK instruction, if the needed storage is less than 32K bytes; or by explicit instructions to push the frame pointer register, transfer to it the stack pointer, and allocate the needed storage, if more than 32K bytes. The frame pointer is either A5 or A6, depending on the -f option used to compile the function (see Section 4.1.2).

2. If any of the registers D2-D7 or A2-A5 are used in the function, a MOVEM.L instruction is used to save their current values on the stack. These registers may be in use by the calling program as register variables; see Section 5.1.6.

3. If the -p option was used to compile the function, a special instruction called a stack probe will be generated: TST.B -144,A7. This instruction is used by Xenix programs to ensure that sufficient memory has been allocated for the run-time stack.

4. If the called function expects to be supplied one or more arguments which are aggregates (structures or unions), it copies the contents of those aggregates to local storage. Note that, when an aggregate is passed by value, the compiler does not push a copy of the aggregate onto the stack. Instead, it supplies the address of the object to the called function, which uses that address to make a local copy of the aggregate.

The offsets of the various components on the stack are indicated by the following diagram.



Function arguments are addressed positively from the frame pointer, while the auto data elements are addressed negatively. Temporaries (used for storage of reusable intermediate expression results) are placed below all of the auto data items but are addressed positively from A7. Only as much storage as is actually needed is allocated for temporaries. Note that the first 32K of automatic data is addressed much more efficiently than any subsequent elements; thus, it is advantageous for functions to use no more than 32K bytes of local storage.

When a function returns to its caller, it first loads the function return value, if any, into predefined registers. The size of the value returned determines the register(s) used:

8 bits	D0.B (low byte of D0, char functions only)
16 bits	D0.W (low word of D0, short functions only)
32 bits	D0.L (all other integral types and pointers)
64 bits	(D0.L, D1.L) (double functions only)

For double precision values, the high order bits are contained in D0. Note that functions returning aggregates (structures or unions) actually return a pointer to a static copy of the aggregate. Because this copy persists only long enough to assign the return value, such functions are nonetheless recursively reentrant (but not multi-tasking reentrant).

After the return value is loaded, the function restores any of the registers D2-D7 or A2-A5 which were saved on entry, by another MOVEM.L instruction. The UNLK instruction is then used to restore both the frame pointer register and the stack pointer, and an RTS is executed to return to the calling function. As is customary in C, that function is responsible for de-allocating any stack space used to push arguments. An ADDQ.L instruction is used if four or eight bytes of arguments were pushed; otherwise, an instruction like

```
LEA  A7,12(A7)
```

is used to restore A7 if more than eight bytes of arguments were pushed on the stack. (Note: LEA is used instead of ADDA.W because it is faster by four clock cycles.)

5.3.4 Assembly Language Interface

Programmers may write assembly language modules for inclusion in C programs, provided that these modules adhere to the object code, linkage, and function call conventions described in the preceding sections. An assembly language module which defines one or more functions to be called from C should observe the following:

(1) The statements defining the functions should be placed in

the text section by preceding them with

```
        SECTION   text
```

This is not an absolute requirement, since the functions will be accessible regardless of the section in which they are defined, but it assures them of placement with the C functions during linking (if the -s option is used). Each function entry must be declared in an XDEF statement:

```
        XDEF      AFUNC
        .
        .
        .
AFUNC   EQU       *
```

(2) If the module is to define data locations to be accessed (using extern declarations) in C modules, those definitions should be placed in the data section by preceding them with

```
        SECTION   data,DATA
```

Each data element must be declared in an XDEF statement in order to be accessible in the C modules:

```
        XDEF      DX,DY,DZ
DX      DC.L      4000H
DY      DC.W      8000H
DZ      DC.L      DX
```

(3) Any of the registers D4-D7 or A2-A6 must be preserved by the module, and the return value loaded into the appropriate data registers.

To call a C function from an assembly language module, an XREF declaration for the function must be included. Before calling the function (via JSR), the caller must supply any expected arguments in the proper order (see Section 5.3.3). After control returns from the called function, the stack pointer must be adjusted by the caller to account for pushed arguments.

```
        XREF      cfunc
        MOVE.L    D0,-(A7)        push argument
        MOVE.L    D1,-(A7)
        JSR       cfunc           call function
        ADDQ      #8,A7           restore stack ptr
```

Data elements defined in a C module may be accessed via XREF statements, as well:

```
        XREF      XD2,XD3
```

```
                    .
                    .
                    MOVE.L    XD2,DØ
```

The following example functions illustrate some of the requirements discussed above.

```
          XDEF      inp,outp
          SECTION   text
*
*   c = inp(ioaddr);      returns byte from specified I/O addres
*  char c;                 returned byte
*  char *ioaddr;           I/O address
*
inp       MOVE.L      4(A7),A0      fetch argument
          MOVE.B      (A0),D0       fetch byte
          RTS
*
*  outp(c.ioaddr);          writes byte to specified I/O address
*  char c;                  byte to be written
*  char *ioaddr;            I/O address
*
outp      MOVE.L     4(A7),A0       fetch first arg (last pushed)
          MOVE.L     8(A7),D0       fetch second arg
          MOVE.B     D0,(A0)        write byte
          RTS
          END
```

SECTION 6:
AmigaDOS System Interface

Although the portable library functions described in Section 3 of this manual define a general purpose interface to the typical environment provided for C programs, there are inevitably many details and variations which are system-dependent. In this section, the execution of C programs and some of the details of the AmigaDOS library implementation are presented in order to clarify the peculiarities of this particular environment.

6.1 Program Execution

Under the AmigaDOS operating system, programs reside in files and may be loaded by user commands or by icon selection. The creation of program files is described in Section 4.2.

6.1.1 Run-Time Structure

Because AmigaDOS is a multitasking operating system, programs are not loaded at any fixed address but are placed in memory at whatever locations are available when the request to load the programs is received. Portions of a program may be loaded at widely different addresses in the default case, although modules compiled with the -s flag described in Section 4.1.2 will be loaded into contiguous regions of memory.

The amount of memory provided by the operating system for the run-time stack is normally 4000 bytes; it can be changed to some other value by the STACK command. See the AmigaDOS User's Manual for details.

6.1.2 Program Execution by Command

When a C program is executed, the function **main** is called to begin execution. Two important services are performed for main before it receives control.

1. The command which executed the program is analyzed, and information from the command line is supplied as parameters to main. The analysis performed and the nature of the parameters supplied will be discussed in detail below. This feature is designed to make it easier to process command line inputs to the program.

2. The buffered text files **stdin** (standard input), **stdout** (standard output), and **stderr** (standard error) are opened and thus available for use by the program. Normally, all three units are assigned to the user's console, but **stdin** and **stdout** may be assigned elsewhere by command line options described below. This feature

allows flexibility in the use of programs which work with text file I/O using the standard getchar and putchar macros.

The simplest way to execute a C program is to type the name of the program file, followed by a return (**<return>**). Since the command line provides a convenient way to supply input to a program, a program execution request will often contain other information. The general format of the command line to execute a C program is:

pgmname [<infile] [>outfile] [args]**<RETURN>**

Required specifiers are shown in emphasized type; optional specifiers are shown enclosed in brackets. Everything after pgmname is optional, as the brackets indicate. The various additional items (<infile and >outfile), if present, must appear before all other command line arguments following the program name. Note that these items do not contribute to the argument count. The term field is used in this section to mean a sequence of printable characters terminated by a white space character, and separated from other fields by one or more white space characters (blanks, tabs, or the end of the command).

pgmname This field names the program to be executed; it is the name of the file created when the program was linked.

<infile The first optional field names a file or device to which the standard input (stdin) is to be assigned. This option is useful only if the program being executed actually uses the standard input (that is, it processes text input using getchar or scanf or makes explicit getc or fscanf calls using stdin). The file or device name must be immediately preceded by a < character. The file must exist, or the program will be aborted with the error message UNABLE TO OPEN REDIRECTION FILE.

>outfile The second optional field names a file or device to which the standard output (stdout) is to be assigned. This option is useful only if the program being executed actually uses the standard output (that is, it generates text output using putchar or printf or makes explicit putc or fprintf calls using stdout). The file or device name must be immediately preceded by a > character. The file is opened as a new file, which discards its previous contents if they already existed and creates an empty file. If the filename specified is invalid or not enough directory space is available to create the new file, the program is aborted with the error message UNABLE TO OPEN REDIRECTION FILE.

args Any additional fields beyond the program name and the two optional fields are extracted and passed to the function main as two arguments:

```
main(argc, argv)
int argc;       /* number of arguments */
char *argv[];   /* array of ptrs to arg strings */
```

Each arg string is terminated by a null byte. On most systems which support C, argv[0] is the name by which the program was invoked. Under AmigaDOS, the program name is not readily available, although all of the other information from the command line is. A dummy argv[0] is therefore supplied (all programs are named c according to argv[0]) but subsequent elements of argv are defined properly. Arguments appear in argv in the same order in which they were found on the command line. Note that the optional redirection specifiers are not included in the argv list of strings.

EXAMPLES

```
CPROG <INPUT.R PQP 12
```

This command executes the program file CPROG, and connects stdin to the file INPUT.R. The main function will be supplied an argc value of 3, with strings c, PQP, and 12 in the argv array.

```
errlst >>errors.log data
```

This command executes ERRLST with stdout connected to ERRORS.LOG, which will be created as a new file. The main function will be supplied with an argc value of 2, with strings c and data in the argv array.

6.1.3 Program Execution by Icon Selection

Under AmigaDOS programs may also be executed by selection of an icon using the Amiga WorkBench interface. Although the main function is supplied with argc and argv, there is no command line associated with the invocation of the program, and the value of argc is always zero. The I/O redirection facility described above is not supported.

6.2 Library Implementation

A complete implementation of the standard C library described in Section 3 is provided for AmigaDOS. The operating system supports a number of powerful features which allow a full implementation of the standard file I/O functions; Section 6.2.1

discusses file I/O. The general topic of device I/O, with emphasis on screen and keyboard, is discussed in Section 6.2.2. Dynamic memory allocation is supported via calls to the operating system, but its operation is not entirely compatible with UNIX implementations, as Section 6.2.3 warns. The basic program entry and exit functions are described in Section 6.2.4, and some special functions unique to the AmigaDOS implementation are discussed briefly in Section 6.2.5.

6.2.1 File I/O

Filenames are specified to the I/O functions according to the standard AmigaDOS format as follows:

dfn:pathname/filename

where dfn: is an optional drive specifier, pathname is an optional directory specifier, and filename is the name of the file. If the drive specifier is omitted, the current drive is used; if the pathname is omitted, the current directory is used. The filename string is terminated by a null byte. Alphabetic characters may be supplied in either upper or lower case; actual filenames use upper case letters only. Only those characters which are legal for filenames under AmigaDOS are acceptable; consult the AmigaDOS documentation for details.

The level 1 I/O functions perform disk I/O by making direct calls to AmigaDOS so that all buffering is performed by the operating system. Programs using the level 2 I/O functions cannot use the rbrk function, because fopen allocates a buffer using getmem.

In the AmigaDOS implementation, both the level 2 (fopen, putc, getc, fclose) and the level 1 (open, creat, read, write, close) I/O functions are limited to 20 open files, including devices, and including the three (stdin, stdout, stderr) which are automatically opened for the main program.

The portable library provides a system-dependent option when a file is opened or created; the programmer may select one of two modes of I/O operation while a file is open. On the AmigaDOS system, there is no difference between the two modes; thus, the text and binary modes are equivalent.

Two other functions should be clarified under the heading of file I/O. The creat function gets a system-dependent argument, the access privilege mode bits; these are ignored under the AmigaDOS implementation. The lseek function has an offset mode, not always implemented, which specifies an offset relative to the end of file. Because AmigaDOS retains the exact file size in its directory, this mode can be and is implemented in this version.

Under AmigaDOS, the external integer location _oserr will contain

the AmigaDOS system error code if a failure indication is obtained from an I/O function. These codes are described in the AmigaDOS reference manual.

6.2.2 Device I/O

The level 1 I/O functions supplied in the Lattice C library for AmigaDOS do not perform any special processing for devices, but simply treat them as if they were disk files. This technique works for many simple devices such as CON:, LPT:, and SER:. In order to perform I/O for special devices, direct calls to the Amiga library functions for the device must be made; consult the AmigaDOS documentation for details.

Screen and keyboard I/O under AmigaDOS is accomplished through windows. Access to the full range of capabilities for windows must be achieved using direct AmigaDOS function calls, but the C library functions can be used to perform simple I/O functions to a window. In particular, stdin, stdout, and stderr are set up to perform I/O to the current window, unless they are redirected as described in Section 6.1.2. Note that AmigaDOS does not support single character input from windows in a straightforward way; thus, when a program requests a single character from a window, no characters will be received by the program until a newline is received from the keyboard. The console I/O functions getch, putch, cgets, cputs, cprintf, and cscanf which are supplied under other implementations of Lattice C are not presently available under AmigaDOS.

6.2.3 Memory Allocation

The full set of memory allocation functions described in Section 3.1 is provided under AmigaDOS, although the sbrk function operates in a slightly different manner from other implementations (see below). The amount of memory in the system available for use via memory allocation depends on the total amount of memory installed and the amount which has been allocated by other active processes.

Under AmigaDOS, the operating system memory allocation facility is used to obtain memory. When direct calls to sbrk are made, a block of the requested size is obtained from AmigaDOS. Unlike other implementations of sbrk, however, the AmigaDOS version of this function does not necessarily return contiguous portions of memory on successive calls. In other words, the UNIX view of sbrk as simply advancing a pointer to the base of a block of memory somewhere outside the program is not correct under AmigaDOS. Note that rbrk is implemented as in other systems, i.e., it returns all allocated memory to the operating system. The AmigaDOS functions called by sbrk and rbrk are AllocMem and FreeMem, respectively.

The level 2 memory allocation function getmem operates as in other implementations, by calling sbrk if none of the available blocks from prior rlsmem calls is large enough to satisfy the requested amount of memory. Instead of passing the exact requested size to sbrk, however, getmem rounds up the needed size according to the value in the external integer _mstep. This approach avoids the high cost of the operating system overhead in processing many small allocation requests from AmigaDOS. If a program primarily makes calls to getmem for large blocks, a large value for _mstep may leave large "holes" in the memory pool. In that case, a small value can be stored in _mstep (this can be done at any time, as often as desired). The default value supplied in the startup module for _mstep is 1024; the value can be over written by including a reference to _mstep in the user's program:

```
extern int _mstep;
```

and then simply making an assignment, such as:

```
_mstep = 512;
```

The level 2 memory allocation functions bldmem, allmem, and rstmem are not available under AmigaDOS. The level 3 functions work as in other implementations, by calling getmem and rlsmem.

Note that the reset function rbrk cannot be used if any of the level 2 I/O functions are also being used on currently open files. Only those files and devices which are being accessed in buffered mode, however, allocate a file access block using getmem; rbrk may be used if the only open file pointers are set up for unbuffered access. A file may be closed, then re-opened after rbrk is called; however, any file pointers must be updated if this is done, because there is no guarantee that the same value will be returned when the file is opened again.

6.2.4 Program Entry/Exit

The start-up module C.O calls _main to begin execution of a C program, and passes to it a copy of the command line which executed the program. Actually, because AmigaDOS does not save the program name portion of the command, the command line passed to _main consists of the characters "c " (lower case 'c' followed by a blank) immediately followed by all of the characters typed after the program name. The standard version of _main supplied in the library analyzes the command line and passes the command-line arguments to main. If the level 2 file pointers stdin, stdout, and stderr are not needed in a program, the file _MAIN.C can be recompiled with a -dTINY option (causing the symbol TINY to be defined), which will eliminate the code which opens those three files, and the resulting object file can be included when the program is linked. Please note the following important

cautions if this is done.

1. The library function printf sends its output to the predefined file pointer stdout, which is normally opened by _main. If the code that performs this function is removed, printf calls will produce no visible output (the I/O library functions ignore attempts to read or write unopened files). A similar caveat applies to the use of scanf, which reads from stdin.

2. If the goal is to avoid including the level 2 I/O functions in the linked program, the library function exit should not be called, since it closes all buffered output file before terminating execution and automatically causes level 2 functions to be included. Call _exit instead.

The program exit functions exit and _exit are described in Section 3.2.4. Under AmigaDOS, the error code argument is passed back to the operating system, where it can be tested in a batch file using the IF command (see AmigaDOS User's Manual).

The source to the standard library version of _main has been supplied as _MAIN.C, which can be customized as needed.

6.2.5 Special Functions

An extensive set of functions which can be called directly from C are available in the library file AMIGA.LIB. The use of these functions is described in the AmigaDOS documentation. Please note the following important caution. Many of the functions in AMIGA.LIB are very similar in name to standard C functions, but often they are quite different in operation. Remember that case is significant in external symbols, so that the function open is not the same as Open. Beware of confusion with Lattice C library functions, and be sure to supply the correct types and number of expected arguments.

APPENDIX A:
Error Messages

This appendix describes the various messages produced by the first and second phases of the compiler. Error messages which begin with the text CXERR are compiler errors which are described in Appendix B.

A.1 Unnumbered Messages

These messages describe error conditions in the environment, rather than errors in the source file due to improper language specifications.

Can't create object file

The second phase of the compiler was unable to create the .O file. This error usually results from a full directory on the output disk.

Can't create quad file

The first phase of the compiler was unable to create the .Q file. This error usually results from a full directory on the output disk.

Can't open quad file

The second phase of the compiler was unable to open the .Q file specified on the LC2 command, usually because it did not exist on the specified (or current) directory.

Can't open source file

The first phase of the compiler was unable to open the .C file specified on the LC1 command, usually because it did not exist on the specified (or current) directory.

Combined output file name too large

The output file name constructed by LC1 or LC2 by combining the source or quad file name with the text specified using the -o option exceeded the maximum file name size of 64 bytes.

File name missing

A file name was not specified on the LC1 or LC2 command.

Intermediate file error

The first phase of the compiler encountered an error when writing to the .Q file. This error usually results from an out-of-space condition on the output disk.

Invalid command line option

An invalid command line option (beginning with a -) was specified on either the LC1 or the LC2 command. See Sections 4.1.1 and 4.1.2 for valid command line options. The option is ignored, but the compilation is not otherwise affected. In other words, this error is not fatal.

Invalid symbol definition

The name attached to -d specifying a symbol to be defined was not a valid C identifier or was followed by text which did not begin with an equal sign.

No functions or data defined

A source file which did not define any functions or data elements was processed by the compiler. This error always terminates execution of the compiler. It can be generated by forgetting to terminate a comment, which then causes the compiler to treat the entire file as a comment.

Not enough memory

This message is generated when either phase of the compiler uses up all the available working memory. The only cure for this error is either to increase the available memory on the system, or (if the maximum is already available) reduce the size and complexity of the source file. Particularly large functions will generate this error regardless of how much memory is available; break the task into smaller functions if this occurs.

Object file error

The second phase of the compiler encountered an error when writing to the .O file. This error usually results from an out-of-space condition on the output disk.

Parameters beyond file name ignored

Additional information was present on the command line beyond the name of the source or quad file to be compiled. The compiler option flags must be specified before the name of the file to be compiled.

Unrecognized -c option

One of the characters following the -c option on LC1 was not a recognized compiler control character. See Section 4.1.1 for a list of the valid compiler control options.

-i option ignored

More than four (4) -i option strings were specified on the LC1 command; only the first four are retained and used.

A.2 Numbered Messages

These error messages describe syntax or specification errors in the source file; they are generated by the first phase of the compiler. A few are warning messages that simply remark on marginally acceptable constructions but do not prevent the creation of the quad file. See Section 4.3.3 for more information about error processing.

1 This error is generated by a variety of conditions in connection with pre-processor commands, including specifying an unrecognized command, failure to include white space between command elements, or use of an illegal pre-processor symbol.

2 The end of an input file was encountered when the compiler expected more data. This may occur on an #include file or the original source file. In many cases, correction of a previous error will eliminate this one.

3 The file name specified on an #include command was not found.

4 An unrecognized element was encountered in the input file that could not be classified as any of the valid lexical constructs (such as an identifier or one of the valid expression operators). This may occur if control characters or other illegal characters were detected in the source file.

5 A pre-processor #define macro was used with the wrong number of arguments.

6 Expansion of a #define macro caused the compiler's line buffer to overflow. This may occur if more than one lengthy macro appeared on a single input line.

7 The maximum extent of #include file nesting was exceeded; the compiler supports #include nesting to a maximum depth of 10.

8 A cast (type conversion) operator was incorrectly specified in an expression.

9 The named identifier was undefined in the context in which it appeared, that is, it had not been previously declared. This message is only generated once; subsequent encounters with the identifier assume that it is of type int (which may cause other errors).

10 An error was detected in the expression following the [character (presumably a subscript expression). This may occur if the expression in brackets was null (not present).

11 The length of a string constant exceeded the maximum allowed by the compiler (256 bytes). This will occur if the closing " (double quote) was omitted in specifying the string.

12 The expression preceding the . (period) or -> structure reference operator was not recognizable by the compiler as a structure or pointer to a structure.

13 An identifier indicating the desired aggregate member was not found following the . (period) or -> operator.

14 The indicated identifier was not a member of the structure or union to which the . (period) or -> referred.

15 The identifier preceding the (function call operator was not implicitly or explicitly declared as a function.

16 A function argument expression specified following the (function call operator was invalid. This may occur if an argument expression was omitted.

17 During expression evaluation, the end of an expression was encountered but more than one operand was still awaiting evaluation. This may occur if an expression contained an incorrectly specified operation.

18 During expression evaluation, the end of an expression was encountered but an operator was still pending evaluation. This may occur if an operand was omitted for a binary operation.

19 The number of opening and closing parentheses in an expression was not equal. This error message may also occur if a macro was poorly specified or improperly used.

20 An expression which did not evaluate to a constant was encountered in a context which required a constant result. This may occur if one of the operators not valid for

constant expressions was present.

21 An identifier declared as a structure or union was encountered in an expression where aggregates are not permitted. Only the direct assignment and conditional operators may be used on aggregates, and explicit or implicit testing of aggregates as a whole is not permitted.

22 (non-fatal warning) An identifier declared as a structure or union appeared as a function argument without the preceding & operator. In Version 3 of Lattice C, aggregates may be passed by value, so that this is a legal construct; the warning message is generated to call attention to the very different meaning of this interpretation from that of previous versions.

23 The conditional operator was used erroneously. This may occur if the ? operator was present but the : was not found when expected.

24 The context of the expression required an operand to be a pointer. This may occur if the expression following * did not evaluate to a pointer.

25 The context of the expression required an operand to be an lvalue. This may occur if the expression following & was not an lvalue, or if the left side of an assignment expression was not an lvalue.

26 The context of the expression required an operand to be arithmetic (not a pointer, function, or aggregate).

27 The context of the expression required an operand to be either arithmetic or a pointer. This may occur for the logical OR and logical AND operators.

28 During expression evaluation, the end of an expression was encountered but not enough operands were available for evaluation. This may occur if a binary operation was improperly specified.

29 An operation was specified which was invalid for pointer operands (such as one of the arithmetic operations other than addition).

30 (non-fatal warning) In an assignment statement defining a value for a pointer variable, the expression on the right side of the = operator did not evaluate to a pointer of the exact same type as the pointer variable being assigned, i.e., it did not point to the same type of object. The warning also occurs when a pointer of any type is assigned to an arithmetic object. Note that the same message may be

a fatal error if generated for an initializer expression.

31 The context of an expression required an operand to be integral, i.e., one of the integer types (char, int, short, unsigned, or long).

32 The expression specifying the type name for a cast (conversion) operation or a sizeof expression was invalid.

33 An attempt was made to attach an initializer expression to a structure, union, or array that was declared auto. Such initializations are expressly disallowed by the language.

34 The expression used to initialize an object was invalid. This may occur for a variety of reasons, including failure to separate elements in an initializer list with commas or specification of an expression which did not evaluate to a constant. Some experimentation may be required in order to determine the exact cause of the error.

35 During processing of an initializer list or a structure or union member declaration list, the compiler expected a closing right brace, but did not find it. This may occur if too many elements were specified in an initializer expression list or if a structure member was improperly declared.

36 A statement within the body of a switch statement was not preceded by a case or default prefix which would allow control to reach that statement. This may occur if a break or return statement is followed by any other statement without an intervening case or default prefix.

37 The specified statement label was encountered more than once during processing of the current function.

38 In a body of compound statements, the number of opening left braces { and closing right braces } was not equal. This may occur if the compiler got "out of phase" due to a previous error.

39 One of the C language reserved words appeared in an invalid context (e.g., as a variable name). Note that entry is reserved although it is not implemented in the compiler.

40 A break statement was detected that was not within the scope of a while, do, for, or switch statement. This may occur due to an error in a preceding statement.

41 A case prefix was encountered outside the scope of a switch statement. This may occur due to an error in a preceding statement.

42 The expression defining a case value did not evaluate to an int constant.

43 A case prefix was encountered which defined a constant value already used in a previous case prefix within the same switch statement.

44 A continue statement was detected that was not within the scope of a while, do, or for loop. This may occur due to an error in a preceding statement.

45 A default prefix was encountered outside the scope of a switch statement. This may occur due to an error in a preceding statement.

46 A default prefix was encountered within the scope of a switch statement in which a preceding default prefix had already been encountered.

47 Following the body of a do statement, the while clause was expected but not found. This may occur due to an error within the body of the do statement.

48 The expression defining the looping condition in a while or do loop was null (not present). Indefinite loops must supply the constant 1, if that is what is intended.

49 An else keyword was detected that was not within the scope of a preceding if statement. This may occur due to an error in a preceding statement.

5Ø A statement label following the goto keyword was expected but not found.

51 The indicated identifier, which appeared in a goto statement as a statement label, was already defined as a variable within the scope of the current function.

52 The expression following the if keyword was null (not present).

53 The expression following the return keyword could not be legally converted to the type of the value returned by the function.

54 The expression defining the value for a switch statement did not define an int value or a value that could be legally converted to int.

55 (non-fatal warning) The statement defining the body of a switch statement did not contain at least one case prefix.

56 The compiler expected but did not find a colon (:). This error message may be generated if a case expression was improperly specified, or if the colon was simply omitted following a label or prefix to a statement.

57 The compiler expected but did not find a semi-colon (;). This error generally means that the compiler completed the processing of an expression but did not find the statement terminator (;). This may occur if too many closing parentheses were included or if an expression was otherwise incorrectly formed. Because the compiler scans through white space to look for the semi-colon, the line number for this error message may be subsequent to the actual line where a semi-colon was needed.

58 A parenthesis required by the syntax of the current statement was expected but was not found (as in a while or for loop). This may occur if the enclosed expression was incorrectly specified, causing the compiler to end the expression early.

59 In processing declarations, the compiler encountered a storage class invalid for that declaration context (such as auto or register for external objects). This may occur if, due to preceding errors, the compiler began processing portions of the body of a function as if they were external definitions.

60 The types of the aggregates involved in an assignment or conditional operation were not exactly the same. This error may also be generated for enum objects.

61 The indicated structure or union tag was not previously defined; that is, the members of the aggregate were unknown. Note that a reference to an undefined tag is permitted if the object being declared is a pointer, but not if it is an actual instance of an aggregate. This message may be issued as a warning after the entire source file has been processed if a pointer was declared with a tag that was never defined.

62 A structure or union tag has been detected in the opposite usage from which it was originally declared (i.e., a tag originally applied to a struct has appeared on an aggregate with the union specifier). The Lattice compiler defines only one class of identifiers for both structure and union tags.

63 The indicated identifier has been declared more than once within the same scope. This error may be generated due to a preceding error, but is generally the result of improper declarations.

64 A declaration of the members of a structure or union did not contain at least one member name.

65 An attempt was made to define a function body when the compiler was not processing external definitions. This may occur if a preceding error caused the compiler to "get out of phase" with respect to declarations in the source file.

66 The expression defining the size of a subscript in an array declaration did not evaluate to a positive int constant. This may also occur if a zero length was specified for an inner (i.e., not the leftmost) subscript of an array object.

67 A declaration specified an illegal object as defined by this version of C. Illegal objects include functions which return arrays and arrays of functions.

68 A structure or union declaration included an object declared as a function. This is illegal, although an aggregate may contain a pointer to a function.

69 The structure or union whose declaration was just processed contains an instance of itself, which is illegal. This may be generated if the * is forgotten on a structure pointer declaration, or if (due to some intertwining of structure definitions) the structure actually contains an instance of itself.

70 The formal parameter of a function was declared illegally as a function.

71 A variable was declared before the opening brace of a function, but it did not appear in the list of formal names enclosed in parentheses following the function name.

72 An external item has been declared with attributes which conflict with a previous declaration. This may occur if a function was used earlier, as an implicit int function, and was then declared as returning some other kind of value. Functions which return a value other than int must be declared before they are used so that the compiler is aware of the type of the function value.

73 In processing the declaration of objects, the compiler expected to find another line of declarations but did not, in fact, find one. This error may be generated if a preceding error caused the compiler to "get out of phase" with respect to declarations.

74 (non-fatal warning) A string constant used as an initializer for a char array defined more characters than

the specified array length. Only as many characters as are needed to define the entire array are taken from the first characters of the string constant.

75 An attempt was made to apply the sizeof operator to a bit field, which is illegal.

76 The compiler expected, but did not find, an opening left brace in the current context. This may occur if the opening brace was omitted on a list of initializer expressions for an aggregate.

77 In processing a declaration, the compiler expected to find an identifier which was to be declared. This may occur if the prefixes to an identifier in a declaration (parentheses and asterisks) are improperly specified, or if a sequence of declarations is listed incorrectly.

78 The indicated statement label was referred to in the most recent function in a goto statement, but no definition of the label was found in that function.

79 (non-fatal warning) More than one identifier within the list for an enumeration type had the same value. While this is not technically an error, it is usually of questionable value.

80 The number of bits specified for a bit field was invalid. Note that the compiler does not accept bit fields which are exactly the length of a machine word (such as 16 on a 16-bit machine); these must be declared as ordinary int or unsigned variables.

81 The current input line contained a reference to a pre-processor symbol which was defined with a circular definition, or loop.

82 The size of an object exceeded the maximum legal size for objects in its storage class; or, the last object declared caused the total size of declared objects for that storage class to exceed that maximum.

83 (non-fatal warning) An indirect pointer reference (usually a subscripted expression) used an address beyond the size of the object used as a base for the address calculation. This generally occurs when an expression makes reference to an element beyond the end of an array.

84 (non-fatal warning) A #define statement was encountered for an already defined symbol. As noted in Section 2.2.1, the second definition takes precedence, but requires an additional #undef statement before the symbol is truly

undefined.

85 (non-fatal warning) The expression specifying the value to be returned by a function was not of the same type as the function itself. The value specified is automatically converted to the appropriate type; the warning merely serves as notification of the conversion. The warning can be eliminated by using a cast operator to force the return value to the function type. This warning is also issued when a return statement with a null expression (i.e., no return value) appears in a function which was not declared void; generation of the warning for this particular context can be disabled using a compile time option (see Section 4.1.1).

86 (non-fatal warning) The types of the formal parameters declared in the actual definition of a function did not agree with those of a preceding declaration of that function with argument type specifiers.

87 (non-fatal warning) The number of function arguments supplied to a function did not agree with the number of arguments in its declaration using argument type specifiers.

88 (non-fatal warning) The type of a function argument expression did not agree with its corresponding type declared in the list of argument type specifiers for that function. Note that the compiler does not automatically convert the expression to the specified type; it merely issues this warning.

89 (non-fatal warning) The type of a constant expression used as a function argument did not agree with its corresponding type declared in the list of argument type specifiers for that function. In this case, the compiler does convert the expression to the expected type.

90 The type specifier for an argument type in a function declaration was incorrectly formed. Argument type specifiers are formed according to the rules for type names in cast operators or sizeof expressions.

91 One of the operands in an expression was of type void; this is expressly disallowed, since void represents no value.

92 (non-fatal warning) An expression statement did not cause either an assignment or a function call to take place. Such a statement serves no useful purpose, and can be eliminated; usually, this error is generated for incorrectly specified expressions in which an assignment operator was omitted or mistyped.

93 (non-fatal warning) An object with local scope was declared but never referenced within that scope. This warning is provided as a convenience to warn of declarations that may no longer be needed (if, for example, the code in which the variable was used was eliminated but not its declaration). It may also occur if the only use of the object is confined to statements which are not compiled because of conditional compilation directives (#if, etc.).

94 (non-fatal warning) An auto variable was used in an expression without having been previously initialized by an assignment statement or appearing in a function argument list with a preceding & (i.e., its address passed to a function). Note that the compiler considers the variable initialized once any statement causes it to be initialized, even though control may not flow from that statement to other subsequent uses of the variable. Note also that this warning will be issued if the third expression in a for statement uses a variable which has not yet been initialized, which may be incorrect if that variable is initialized inside the body of the for statement.

APPENDIX B:
Compiler Errors

This appendix describes the procedure to be used for reporting compiler errors. These are errors that result not from the user's incorrect specifications but from the compiler itself failing to operate properly. There are five general kinds of errors which can occur:

1. The compiler generates an error message for a source module which is actually correct.

2. The compiler fails to generate an error message for an incorrect source module.

3. The compiler detects an internal error condition and generates an error message of the form

   CXERR: nn

   where nn is an internal error number.

4. The compiler dies mysteriously (crashes) while compiling a source module.

5. The compiler generates incorrect code for a correct source module.

The last type of error is, of course, the most difficult to determine and the most vexing for the programmer, who has no indication that anything is wrong until something inexplicably doesn't work, and who only concludes that the compiler is at fault after a long and painstaking study of his or her own code.

Lattice is anxious to know about and repair any compiler errors as quickly as possible, so please report any problems promptly. The difficulties encountered may be spared the next programmer if this is done. In order to maintain a more precise record of the bugs that are discovered, all problems should be reported in writing to:

Lattice, Inc.
P.O. Box 3Ø72
Glen Ellyn, Illinois
6Ø138

In all cases, include the following items of information:

1. A listing of the source module for which the error occurred. Don't forget to include listings of any #include files used (and watch out for #include file

nesting; don't forget the inner files as well). Supplying the source on IBM PC-compatible disk format will save time.

2. The revision number of the compiler, when it was purchased and the serial number.

3. Your name and address and, if possible, a telephone number with information about the best time to call.

4. A description of the problem, along with any other information which may be helpful such as the results of your investigation into the problem. Obviously, errors of type 3 (see above) don't need anything more than a terse "Causes CXERR 23."

Our current policy calls for a free update to the first finder of a compiler bug.

APPENDIX C:
List of Files

The following files are supplied as part of the AmigaDOS 68000 compiler package:

Executable Files

LC	Compiler command line handler
LC1	C compiler (phase 1)
LC2	C compiler (phase 2)
OMD	Object module disassembler

Run-time and Library Files

C.O	C program entry/exit module for AmigaDOS
LC.LIB	Run-time and I/O library

C Source Files

_MAIN.C	Standard library version of _main

C Header Files

STDIO.H	Standard I/O header file
CTYPE.H	Character type macros header file
ERROR.H	Header file defining UNIX error numbers
FCNTL.H	Header file defining level 1 I/O codes
IOS1.H	Header file defining level 1 I/O structures
DOS.H	Environment information header file
MATH.H	Mathematical functions header file
LIMITS.H	Defines limiting values for math functions
SETJMP.H	Defines save structure for setjmp and longjmp

Assembly Language Source Files

C.A	C program entry/exit module

Commodore Business Machines, Inc.
1200 Wilson Drive, West Chester, PA 19380

Commodore Business Machines, Limited
3370 Pharmacy Avenue, Agincourt, Ontario, M1W 2K4